University of Alberta

EVOLUTIONARY OPTIMIZATION TECHNIQUES AND RECONFIGURABLE HARDWARE

by

Madhura Purnaprajna

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Electrical and Computer Engineering

Edmonton, Alberta
Spring 2005

0-494-08159-7

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file *Votre référence*
ISBN:
Our file *Notre reference*
ISBN:

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

Poojyaaya Raghavendraya Sathya Dharma Ratayacha
Bhajathaam Kalpavrukshaaya Namataam Kaamadhenuve

Abstract

An optimization scheme for embedded system design using hardware and software components is presented. The goal of suitably partitioning the system into hardware and software is achieved using *Genetic Algorithms* (GA). The optimization objective is to reduce the time taken and the power consumed during task execution. The suitability of introducing a reconfigurable hardware resource over preconfigured hardware is explored for the same objectives. Further, the procedure to allocate optimal number of resources based on the design objective is proposed. A test environment is developed using randomly generated task graphs.

In applying evolutionary optimization techniques to reconfigurable architectures, the design of a run time reconfigurable *Fuzzy Logic Controller* (FLC) is presented. Four design strategies of implementing the FLC are presented, which includes a feasibility analysis for the maximum design size. Further, an implementation framework for hardware-software co-design and a self-reconfiguring Fuzzy Logic Controller is proposed.

Acknowledgements

My sincere thanks to my Supervisors Dr. Marek Reformat and Dr. Witold Pedrycz for their support and guidance. I would also like to extend my gratitude to all my friends for their help throughout my Master's program.

Table of Contents

1 Introduction 1
- 1.1 Motivation . . . 1
  - 1.1.1 Optimization . . . 2
  - 1.1.2 Adaptability . . . 3
- 1.2 Approach . . . 3
- 1.3 Thesis Outline . . . 4
- 1.4 Design Environment . . . 4

2 Background 5
- 2.1 Hardware Software Partitioning . . . 5
  - 2.1.1 Prior Work in Design Partitioning . . . 5
  - 2.1.2 Scheduling and Allocation . . . 6
  - 2.1.3 Co-design System Architecture . . . 8
- 2.2 Optimization Framework: Genetic Algorithms . . . 9
  - 2.2.1 Phenotype to Genotype . . . 9
  - 2.2.2 Initial Population . . . 10
  - 2.2.3 Evaluation Function . . . 10
  - 2.2.4 Reproduction . . . 11
  - 2.2.5 Termination Criteria . . . 11
- 2.3 Fuzzy Architectures . . . 11
  - 2.3.1 Brief Introduction of Fuzzy Set Theory . . . 12
  - 2.3.2 Fuzzy Logic Controller . . . 12
  - 2.3.3 Fuzzifier . . . 12
  - 2.3.4 Inference Scheme . . . 13
  - 2.3.5 Defuzzifier . . . 15
- 2.4 Review of Fuzzy Logic Controllers . . . 16
  - 2.4.1 Memory based Fuzzy Logic Controller Implementations . . . 17
  - 2.4.2 Memory-less Fuzzy Logic Controller Architectures . . . 18
- 2.5 FPGA architectures . . . 18
  - 2.5.1 Virtex-II Architecture . . . 19
  - 2.5.2 Virtex-II Pro . . . 20
  - 2.5.3 Virtex-IV . . . 21
  - 2.5.4 Microblaze Processor Architecture . . . 21

2.5.5 Cell Matrix 22

3 Evolutionary Optimization based Partitioning: Approach 25
3.1 Generic System Model 25
3.2 Problem Description 25
3.3 Chromosomal Representation 26
3.4 Fitness Function 26
3.4.1 Scheduling 28
3.4.2 Time Optimization 29
3.4.3 Power Optimization 30
3.4.4 Time and Power Optimization 30
3.4.5 Optimization for Resource Allocation 31
3.5 Details of GA parameters 33
3.5.1 Initial Population 33
3.5.2 Reproduction 33
3.5.3 Termination Criteria 33

4 Evolutionary Optimization based Partitioning: Results 34
4.1 Benchmarking Scenario 34
4.1.1 Task Graphs 34
4.1.2 Design Parameters 36
4.2 GA Parameters 37
4.3 Experiments 37
4.3.1 Experiment 1 37
4.3.2 Experiment 2 38
4.4 Results 38
4.4.1 Random Search and GA 38
4.4.2 Results from Experiment 1 39
4.4.3 Results from Experiment 2 43
4.5 Observation and Inference 45

5 Fuzzy Logic Controller: FPGA Implementation 49
5.1 Hardware Architecture 49
5.1.1 Fuzzification 49
5.1.2 Rule Evaluation 52
5.1.3 Defuzzifier 55
5.2 Software Architecture 56
5.2.1 IO configuration 57
5.2.2 Hardware-Software Architecture 57
5.3 Feasibility in Virtex-II: Hardware Architecture 58
5.3.1 FLC : Combinatorial Rule Base 59
5.3.2 FLC: Memory-based Rule Base 59
5.3.3 Resource Utilization 60
5.3.4 Frequency of Operation 61

5.4 Observation and Inference 61

6 Reconfigurable Fuzzy Logic Controller **63**
6.1 Architectural Changes 63
6.1.1 System Inputs 63
6.1.2 Input and Output Membership functions 64
6.1.3 Defuzzification 64
6.2 FPGA Configuration Mechanisms 64
6.2.1 Configuration Modes 64
6.2.2 Reconfiguration 65
6.3 Partial Run time Reconfiguration in Xilinx FPGAs 66
6.3.1 Module Based 66
6.3.2 Difference Based 67
6.4 Reconfigurable Fuzzy Logic Controller on an FPGA 67
6.4.1 Module Based: Configuration 1 67
6.4.2 Module Based: Configuration 2 68
6.4.3 Module Based: Configuration 3 68
6.4.4 Difference Based Reconfiguration 69
6.5 Reconfiguration Schemes 69
6.5.1 System ACE - CompactFlash Solution 69
6.5.2 Internal Configuration Access Port 71
6.6 Design and Implementation on Xilinx Multimedia Board 72
6.6.1 Default Configuration 72
6.6.2 Dynamic Reconfiguration 72
6.7 Observation and Inference 73

7 Conclusions and Future Work **76**
7.1 Conclusions 76
7.1.1 Modeling 76
7.1.2 Partitioning and Interfacing 76
7.1.3 Validation 77
7.1.4 Implementation 77
7.2 Future Work 78
7.2.1 Improvements in Hardware Software Partitioning 78
7.2.2 Reconfigurable Fuzzy Logic Controllers: Design Alternatives 79
7.2.3 Hardware Software Co-Simulation 80
7.2.4 Reconfigurable Processor Architecture 80
7.2.5 Evolvable Hardware 81

Bibliography **82**

A Parameters to generate TGFF **87**

B Parameters for Power and Time calculations **88**

List of Tables

1.1 Hardware Design Components 2

4.1 Results for Experiment-1 Case 1: Time 39
4.2 Results for Experiment-1 Case 1: Power 40
4.3 Results for Experiment-1 Case 1: Time & Power 41
4.4 Results for Experiment-1 Case 2: Time 41
4.5 Results for Experiment-1 Case 2: Power 41
4.6 Results for Experiment-1 Case 2: Time & Power 42
4.7 Power Optimization with Hard Deadline 44
4.8 Simultaneous Time and Power Optimization with Hard Deadline . . 45
4.9 Time Optimization with Power Limitation 48
4.10 Simultaneous Time and Power Optimization with Power Limitation 48

5.1 Rule Base in Virtex-II 59
5.2 FPGA Resource Utilization 60
5.3 Maximum Frequency of Operation 61

6.1 Configuration Modes 65
6.2 Configuration Modes 66

B.1 Parameters for Time calculations 88
B.2 Parameters for Power calculations 88

List of Figures

1.1 Device Trends 2

2.1 Algorithm for GA Implementation 10
2.2 Block diagram of Fuzzy Logic Controller 13
2.3 Triangular Membership functions 14
2.4 Trapezoidal Membership functions 14
2.5 Rule Matrix 15
2.6 Method of Inference: Min-Max 16
2.7 Method of Defuzzification 16
2.8 Virtex-II Architecture, [1] 19
2.9 Configurable Logic Block, [1] 20
2.10 Virtex-II Slice, [1] 21
2.11 Virtex-II Pro, [2] 22
2.12 Microblaze Architecture, [3] 23
2.13 CellMatrix Architecture, [4] 24

3.1 Hardware-Software System Architecture 26
3.2 GA applied to Partitioning 27
3.3 Chromosome 1 27
3.4 Chromosome 2 28
3.5 Chromosome for Resource Allocation 32

4.1 Task Graph with 200 nodes 35
4.2 Results of GA v/s Random Search 39
4.3 Variation in the fitness function over generations 40
4.4 Optimization for Time 42
4.5 Optimization for Power 43
4.6 Task distribution of tasks on Hardware / Software - Case 1 44
4.7 Task distribution of tasks on Hardware / Software - Case 2 45
4.8 Distribution of Hardware / Software utilization in system implementation 46
4.9 Distribution of Hardware (Preconfigured and Reconfigurable) / Software in system implementation 47

5.1 Triangular Membership functions 50

5.2 Trapezoidal Membership functions 51
5.3 Schematic for Fuzzification 52
5.4 Detailed Schematic for Fuzzification 53
5.5 Schematic for Inference Scheme 54
5.6 Memory Structure 55
5.7 Schematic for Defuzzifier: Centre of Gravity 56
5.8 FLC with GPIOs 58
5.9 Feasibility: Combinatorial Implementation 59
5.10 Feasibility: Memory based Implementation 60

6.1 Layout of Partially Reconfigurable Modules, [5] 67
6.2 Configuration 1 68
6.3 Configuration 2 69
6.4 Configuration 3 70
6.5 SystemACE environment,[6] 71
6.6 MPU setup, Configure from CompactFlash, [6] 73
6.7 Configuration Change 74
6.8 Sequence of configuration 75
6.9 Partial Reconfiguration 75

7.1 Two-level GA 78
7.2 Partial Runtime Reconfiguration 79
7.3 Design Alternatives 80
7.4 GA Machine 81

C.1 Modified Architecture 89
C.2 Shared Bus Architecture 90

Chapter 1

Introduction

System design involves intelligent management of available resources to meet performance requirements, while achieving the desired functionality. The architecture of the system is governed by a trade-off between a number of performance parameters such as time, area, weight, cost, power, reliability and flexibility. Specifically, embedded system design necessitates the use of time and power efficient designs. In addition to these requirements the flexibility and adaptability of the architecture in terms of time to design, configure and reconfigure is also a major deciding factor.

1.1 Motivation

Programmable logic devices have matured from one time, off-line programmable to runtime programmable, further to complex configurable system on chip architectures. These complex reconfigurable devices allow embedded processors and microcontrollers to reside alongside programmable hardware resources on the same chip. Such devices encourage designers to explore applications targeted towards programmable system on chip. In this thesis we focus on the issue of optimality and flexibility in a heterogeneous design architecture comprising of hardware and software elements.

Designs traditionally use one of the devices shown in Figure 1.1, restricting its flexibility and speed to the device characteristics. *Application Specific Integrated circuits* (ASIC), are customized chips for dedicated applications with very little programmability. *Field Programmable Gate Arrays* (FPGAs) are programmable structures which can be infinitely reprogrammed, but do not provide the performance benefits as compared to ASICs. *General Purpose Processors* (GPPs) are more suited for general purpose computing and are easily programmable.

The quest of self-adaptive devices and device architectures, has led to the approach of 'Evolutionary Circuit Design' and 'Evolvable Hardware' focusing on the static and dynamic design of electronic circuits. The application of evolutionary techniques to the field of hardware design aims at evolving the architecture. In this thesis we focus on Digital Evolvable Hardware, mainly in applying evolution-

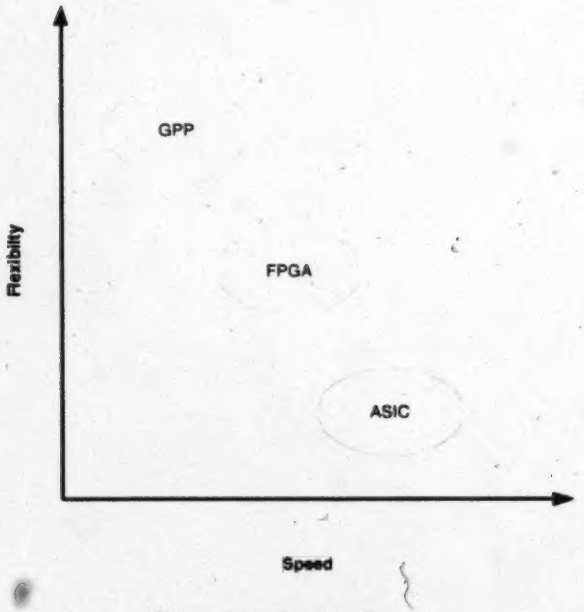


Figure 1.1: Device Trends

ary optimization techniques in evolving FPGA based designs. Table 1.1 lists the existing devices and design environments available for electronic system design.

Table 1.1: Hardware Design Components

Device	Application	Environment
ASIC	Custom designs	Static
FPSOC	Field Programmable System on Chip	Dynamic
FPGA	Field Programmable Gate Device	Dynamic
FPAA	Field Programmable Analog Arrays	Dynamic
FPTA	Multicellular System	Dynamic
PLA	Digital Hardware	Static
CellMatrix	Self-Configurable Hardware	Dynamic

1.1.1 Optimization

Systems are graded in terms of their performance/watt ratio. Low power design strategies are essential to prolong battery life, reduce packaging and cooling costs for mobile applications and operations at high speeds. In conventional microprocessors a high-speed operation often results in high power consumption, which is undesirable. We look at an architecture made up of hardware and software elements in a co-design framework, which addresses the issues of performance and power

management. The optimization process involved in Hardware-Software co-design may be broadly divided into two steps.

- To determine the number and type of resources required for a given design objective
- To map the tasks into hardware and software resources such that the new design meets the desired functionality while achieving optimization objective.

1.1.2 Adaptability

A flexible design has the capability of adapting itself to a number of factors such as system requirements, design changes and a change in resource availability. Designs that are infeasible on account of resource requirement can be accommodated by sharing the same area and reconfiguring parts of the device that is unused. The adaptive nature of devices allows more than one design to reside at the same location. In this scenario, the time to adapt is of major relevance. Fast adaptive design architectures reduce the complexity of the system and allow a single architecture block to be multi-functional. We take a look at reconfigurable architectures as a method of building adaptive and evolvable hardware. Our focus is on the use of fast reconfigurable hardware structures as a replacement to multiple preconfigured fixed structures.

1.2 Approach

In this thesis we address the architectural issues through two different perspectives. First, we look at Hardware-Software co-design as a possible efficient design approach. We define a general architectural model, and identify the possible methods of optimizing such architecture. The system architecture is defined in terms of a number of system parameters. The system architecture can be modified by manipulating these parameters. Our objective is to optimize the system for both power and time considerations. Area and cost restrictions are reflected in terms of resource constraints. The simultaneous optimization of power and time is addressed, which has not been widely discussed previously. We propose a scheduling algorithm that manages simultaneous scheduling and allocation based on task priority. Cost functions reflecting the design objectives are defined. These functions can be altered as per design objectives. A test environment has been developed to analyze and validate these optimization objectives.

To demonstrate our objectives we look at an adaptive system with devices that are runtime reconfigurable to address the issue of flexibility and re-usability. As an example, we look at the design and implementation of an adaptive Fuzzy Logic controller. Finally, the architecture of a Self-reconfigurable Fuzzy Logic controller, built on a hardware-software co-design platform is proposed.

1.3 Thesis Outline

Chapter 2 co-relates the optimization approach in hardware software partitioning to other scheduling techniques that exist in literature. First, a review of existing approaches to solve the problem of partitioning and an introduction to our optimization framework is looked into in this chapter. After which, an introduction to Fuzzy Logic and Fuzzy Logic Controllers is followed by a review of existing implementations in Fuzzy Logic Controllers. Finally, a section on the existing reconfigurable hardware architectures is provided.

Chapter 3 details our approach of solving the partitioning issue and Chapter 4 discusses the experimental framework and the results obtained.

Chapter 5 discusses the software and hardware architectural issues involved in implementing a Fuzzy Logic Controller on an FPGA. The methods of reconfiguration and our approach to arrive at a configuration solution are discussed in Chapter 6.

Finally, conclusions are drawn in Chapter 7. An area of future research, which includes a design solution for an adaptive Fuzzy Logic controller in a co-design environment, is proposed. This approach acts as the bridge for the two objectives of optimality and flexibility.

1.4 Design Environment

Following is the design environment for the experimental setup.

- FORTE Developer: Blade1000
- Xilinx
  - Virtex-II Multimedia Board
  - ISE 6.2
  - Embedded Design Kit 6.2
- ModelSim 5.4SE
- Windows Platform from CMC

Chapter 2

Background

In this chapter, a review of existing optimization techniques and their direct relevance to the problem of hardware software partitioning is presented. The first section of this chapter addresses the design considerations in a Hardware-Software co-design environment. This section includes a review on the approaches considered by other researchers in addressing the issue. We have addressed the problem of Hardware-Software partitioning using Genetic Algorithms. The fundamental concepts of Genetic Algorithms are reviewed in the next section.

The third section of the chapter provides a brief introduction on Fuzzy Set Theory and elaborates the fundamentals of a *Fuzzy Logic Controller* (FLC). This is followed by a review and classification of existing FLCs. A review of the architectural description of existing Xilinx FPGAs is included in the final section. A short introduction to the concept of 'Cell Matrix' is also included as a preview to future prospective devices for Evolvable hardware.

2.1 Hardware Software Partitioning

Partitioning in Hardware-Software co-design is the process of distributing the tasks in an application onto the existing hardware and software resources.

2.1.1 Prior Work in Design Partitioning

A number of heuristics have been explored to optimize the process of design partitioning in hardware software co-design [7], [8], [9]. Many of these heuristics aim at multi-objective optimization, with conflicting objectives for hardware and software design units [10], [11], [12], [13], [14]. A comparison to these existing techniques is difficult due to the absence of well-defined benchmarks in terms of architecture and functionality, for co-designs. However, the concept of replicable randomly generated task-graphs [15], used here, encourages future comparisons. The test graphs described represent the system at task-level granularity.

In [9] the functional partitioning in hardware software co-design is carried out using three heuristic search algorithms: Simulated Annealing, Tabu Search and Genetic Algorithms. A shared bus architecture with an ASIC (or an FPGA without reconfiguration capabilities) and a soft processor is considered for the co-design environment. The objective is to optimize the design for speed, when subject to area constraints. The comparative study, suggest that Tabu Search is optimal in terms of computation time and the quality of the result. In [16], the multi-objective criterion of minimizing time and the energy-delay product is addressed. Another study [12] compares the partitioning of knowledge based systems and circuit-partitioning using Simulated Annealing, Kernighan & Lin and Hierarchical clustering. With the introduction of hardware elements in an all-software implementation, an observation of an overall improvement in the speed of operation is made [13]. This is achieved by moving repetitive operations, such as loops in a task graph on to hardware.

In [17], reconfigurable hardware resources with finite reconfiguration times have been introduced in the hardware architecture. Genetic Algorithms is used to optimize the speed of application. The fitness function measures the speed improvement achieved using hardware-software co-design over an all-software implementation. The advantage of introducing partially reconfigurable resources in hardware to enhance resource utilization has also been used in [16] and [18].

Predominantly, most power optimization techniques focus on fabrication-related issues, such as voltage scaling [19], reducing clock speeds and introducing gated clocks [20]. *Multi-objective GA for Hardware-Software Co-synthesis* (MOGAC) [10] aims at multi-objective partitioning criteria for optimization in power and costs, with time constraints. Power calculation is based on predefined values for peak power dissipation, idle power consumption and power efficiency for each hardware resource. This calculation is based on obtaining energy and ignores the additional power consumed due to the switching activity between them. Also, the GA formulation requires a complicated technique of clustering and validation to obtain only valid structures. MOGAC does not assume any limitation in the number of hardware or software elements in its architecture. This is quite contradictory to practical applications.

2.1.2 Scheduling and Allocation

The process of partitioning involves scheduling and allocation of heterogeneous resources.

Scheduling determines the exact time instance at which a task is executed. It is classified on the basis of its constraints, such as time and resources. In time constrained scheduling, bound by a hard-deadline, the algorithm computes the optimal combination of resources required to meet this timing requirement. In resource constrained scheduling, with a restriction in the number of resources available for the task computation, scheduling decides the best time instance to execute a task,

to minimize the total computation time. The *As Soon As Possible* (ASAP) and the *As Late As Possible* (ALAP) algorithms compute task schedules for unconstrained graphs. They are primarily used to determine the upper and lower bounds for the schedule times [21].

Allocation is the procedure of determining the task to resource mapping, i.e., it decides the assignment of a task to a resource. Optimization procedures are adapted in allocation to maximize the utilization ratios of the resources.

2.1.2.1 Applications of Scheduling and Allocation

A similar approach of scheduling and allocation has been addressed in operations research, heterogeneous computing, multi-processor scheduling and high-level synthesis.

Heterogeneous computing employs a collection of machines with varying architectures. Each of these machines are optimized to perform computationally diverse applications. The scheduling procedure performs a task to machine mapping for the sequence of tasks with the intention of minimizing time, cost or both. A comparative study is made using eleven different heuristics to optimize the system implementation in [22]. This concept closely resembles the approach of mapping a task to a resource in Hardware-Software partitioning.

Operations Research addresses scheduling in the classical example of the *Traveling salesman's problem* (TSP). Here, the optimization involves minimizing the total time taken by the salesman. This problem of scheduling arrives at a sequence which takes the least amount of time for execution. There are no fixed constraints of precedence relation, which makes it different from the problem addressed here. Apart from this, there are three shop scheduling techniques which resemble partitioning and scheduling. They are open shop, job shop and flow shop scheduling. All these techniques aim at minimizing the total make span of the schedule. Open shop scheduling, like TSP, does not consider any predefined sequence of operation. Whereas, job-shop and flow shop scheduling have a predefined sequence of operations to be scheduled. Flow shop scheduling is a special case of job shop scheduling with a fixed task to resource mapping. The objective is to arrive at a sequence of operation which reduces the total makespan, [23]. The absence of heterogeneous resources and dissimilar tasks makes these shop scheduling problems different from partitioning and scheduling in co-design.

In high-level synthesis, a register-transfer level netlist is derived from the behavioral or structural hardware description [24]. This netlist is then scheduled and allocated based on the number and type of available hardware resources. This problem of scheduling in high-level synthesis resembles the case of multi-processor scheduling, which is a well known NP-Complete problem [25].

2.1.3 Co-design System Architecture

The computational sequence of tasks in an embedded system are represented by *Control and Data flow graphs* (CDFG). For a *System on a Chip* (SoC) application, these tasks can be distributed both on hardware as well as software resources. Resources termed as hardware can be *Application Specific Integrated Circuits* (ASIC) or a reconfigurable device, such as a *Field Programmable Gate Array* (FPGA). The devices commonly referred to as 'software' are *General-Purpose Processors* (GPP) or *Digital Signal Processors* (DSP). On account of customization in hardware, ASICs have a very large design time, but have a significantly high performance. However, ASICs are not always suitable due to their enormous design times and lack of reconfiguration capability. FPGAs, on the other hand, are readily reconfigurable devices and hence chosen as 'hardware' in our architecture. These devices play a major role in co-design in order to enhance the resource utilization of the system with their reconfigurable capabilities. With the advent of runtime reconfiguration, FPGAs provide the ability to adapt to the design requirement. This stands as a significant advantage compared to customized ASICs. Although they are time-efficient, FPGAs are expensive in terms of area and power. Also, introduction of reconfiguration adds time and power overheads. In the absence of runtime reconfiguration, the FPGA architecture is limited to its pre-defined fixed configuration. FPGAs provide task execution at higher speeds than most GPPs, but are not power-efficient. GPPs are relatively less time-efficient but provide a low-cost, low power additional resource to explore parallelism, as compared to FPGAs. Also, they are not restricted by their configuration options since there is no reconfiguration overhead. A slow speed, hence low power GPP is chosen as 'software' in the architecture under consideration.

Allocating all the tasks on the same hardware resource for speed enhancement complicates the scheduling procedures. Similarly, forcing all the tasks on software drastically overshoots the time deadline. Hence, for a time and power optimal applications, it is efficient to utilize parallelism by splitting the design on to multiple processing elements (FPGAs and GPPs). The advantages of introducing FPGAs to enhance the speedup has been detailed in [26].

2.1.3.1 Hardware and Software characteristics

There exists a fundamental difference in the task execution by the two types of resources. Number of tasks that can be executed concurrently on an FPGA depends on the number of system units that can be configured on the device. This is not applicable in software, since tasks mapped to software are executed sequentially. Due to the absence of concurrency in software, the number of tasks that can be executed at any given time is restricted to the number of GPPs available. Consequently, with respect to execution times, it is assumed that a task implemented in software takes greater number of clock cycles than that on hardware. Partial run-time reconfiguration on FPGAs, allows part of the device to be reconfigured while rest of

the device functions normally [5]. Introduction of partial run-time reconfigurable capabilities increases resource utilization by allowing reconfiguration of the unused resources to adapt according to the system resource requirement, encountered during scheduling. However, reconfiguration involves an additional cost of time and power.

In time constrained scheduling, with a set hard-deadline, the resources required to satisfy these timing constraints constitute the target architecture for the system implementation. Resource allocation and utilization has been only partially addressed in the case of Hardware-Software co-design. In [27], experiments are carried out to compare the improvement in system performance by increasing the number of dynamically reconfigurable logic elements and varying the reconfiguration time. However, there is no established method for determining the optimal number of resources required for a given system based on the design objective. Most co-design approaches consider resource-constrained architectures. Such architectures may have under-utilized resources which add to idle power and unused area in turn affecting the system efficiency.

In our discussion, for time optimization the quality of partition is determined by the total time required for task execution based on the cumulative latencies at the nodes. The additional overheads account for the inherent delays incurred at edges of the graph. For power, it is the summation of the power consumed by each of the graph nodes. In addition, the inter-processor communication overheads add to the switching activity in the system. These overheads are incurred only in heterogeneous multi-processor systems.

2.2 Optimization Framework: Genetic Algorithms

GA is a powerful and widely used stochastic search based algorithm [28]. It is an effective technique to solve combinatorial optimization problems, which are known to be non-deterministic in nature and are associated with a large combination of feasible solution space or search space. It has been demonstrated that GA is effective in avoiding the local optimal solution and achieves results close to the global optima.

Genetic algorithms are based on the theory of evolution. Unlike other approaches to optimization, GA encodes the design parameters to arrive at the best solution, than actually manipulating the parameter. In contrast to evaluating a single point, GA works on a set of solutions and evaluates them based on a predefined objective. The evaluations are based on a probabilistic approach rather than a deterministic rule. Figure 2.1 shows the logical flow for the implementation of GAs, [29]. The key issues in the GA formulation are described in the following sections.

2.2.1 Phenotype to Genotype

Representation is a key issue in Genetic Algorithms. As the DNA structure reflects the individual physical characteristics, the physical traits of the solution are replaced

by an internally coded representation. The possible solutions to the problem under consideration are encoded as a string of finite length, referred to as a chromosome. Each element of the chromosome is called an allele. The allele could be represented as a real or a binary number. When used with real values, it is termed as *Real Coded GAs* (RCGAs). Real Coded GAs are used primarily in continuous domain optimization problems. Binary coded GAs have been very widely used in literature.

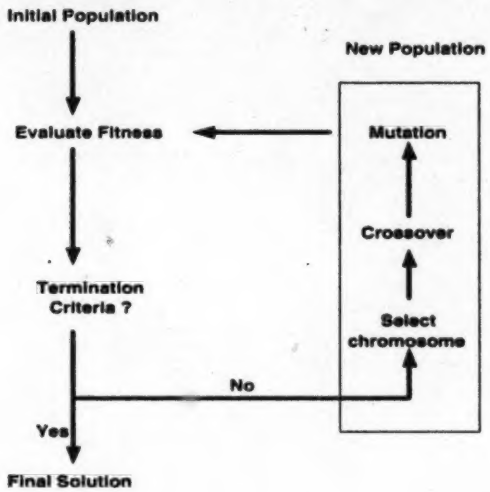


Figure 2.1: Algorithm for GA Implementation

2.2.2 Initial Population

GA works on processing probable set of solutions. This first set of potential solutions is called the initial population. This set of solution is randomly initialized for diversity in the search space. These diverse set of solutions are then fine tuned on the basis of probabilistic evaluation and application of the genetic operators.

2.2.3 Evaluation Function

The quality of each chromosome is assessed by an evaluation function. The evaluation based on the desired objective is termed as the 'fitness' of the chromosomes. Often, the fitness function has to be scaled to enable selection of individuals in the entire population with selective discrimination. In other words, a wide difference in the best and worst case fitness values in a given generation may lead to elimination of a large set of chromosomes with good genes but lower fitness values.

2.2.4 Reproduction

Based on the fitness of the individual, the next generations of possible solutions are created in the process of reproduction. Stochastic transformations are introduced to form new individuals. This probabilistic approach creates a wide space of possible solutions. Adhering to the theory of the survival of the fittest, the process of reproduction retains the 'good' individuals and tried to suppress the 'not so good' set of solutions.

2.2.4.1 Selection

The two common techniques of selection are the Roulette wheel and the Tournament selection criteria. In the Roulette Wheel technique the probability of selecting an individual is directly proportional to its fitness function. The tournament selection criterion picks a few chromosomes and then selects the winner among these chromosomes based on the best fitness value. In addition, the *elitist criterion* selectively retains the best solution in every generation. This ensures that the best results are not lost in the process of reproduction.

2.2.4.2 Modification

Two main operators for modifying the population after selection are crossover and mutation. *Crossover* creates new individuals by copying parts of two other individuals. *Mutation* introduces random transformations to the existing chromosome and creates a new individual. Mutation operation widens the search space, and avoids restriction of the solution to the local minima. These operations are probabilistic and are decided based on parameters termed as *probability of crossover* (P_c) and *probability of mutation* (P_m).

2.2.5 Termination Criteria

The algorithm terminates when the fitness function stabilizes after iterating over a predetermined number of generations. The end point of the algorithm is determined based on two frequently used techniques. The first uses a fixed number of generations (usually fixed by experimenting), after which the result so obtained is the near optimal result. In the second method, the iterative computation continues till the variation in the best value of the solution, between two consecutive iterations, is very little or almost zero.

2.3 Fuzzy Architectures

The concept of Fuzzy Set Theory was introduced by L. Zadeh in 1965, as a means of generalizing the classical set theory. This theory of applying fuzzy linguistic

description to process control has resulted in the design of fuzzy logic based controllers. These controllers provide a mechanism to describe complex structures in terms of simple linguistic variables instead of the complex mathematical process model. On account of the simplicity and ease of implementation, these controllers have been used in a variety of applications. Fuzzy logic controllers have been used in the design of *proportional-integral-derivative* (PID) controllers for adaptive control and optimization of the controller gains.

2.3.1 Brief Introduction of Fuzzy Set Theory

Classical set theory defines a fixed Boolean relationship to determine the associativity of an element to a set. For example, an element x can have the following membership with respect to the set A, [30]

$$\chi_A(x) = \begin{cases} 1, & \text{iff x} \in \text{A} \\ 0 & \text{iff x} \notin \text{A} \end{cases} \tag{2.1}$$

This may be represented as,

$$\chi_A(x) : X \rightarrow \{0,1\} \tag{2.2}$$

In contrast to the classical set theory, the Fuzzy set theory associates more than one degrees of membership. It maps every element of the universe of discourse X to the interval [0,1] , which can be expressed as,

$$\mu_A(x) : X \rightarrow [0,1] \tag{2.3}$$

For a system, fuzzy set theory associates the antecedents (inputs) to the consequents (outputs) based on a fixed set of rules, called the inference scheme.

2.3.2 Fuzzy Logic Controller

The generic structure of a N-input 1-output Fuzzy Logic Controller (FLC) is shown in Figure 2.2. The functionality may be divided into three basic building blocks: the Fuzzifier, the Inference Scheme and the Defuzzifier. The functional description of each of these blocks is detailed in the following sections.

2.3.3 Fuzzifier

Fuzzification is the process of mapping the real-world (crisp) inputs to fuzzy sets based on the input membership functions. The commonly used membership functions are triangular, trapezoidal or Gaussian in nature.

Figure 2.3 and Figure 2.4 shows the membership function with an overlap of two and the degree of association represented as μ which lies between 0 and 1. Every input is associated with one or more fuzzy sets (membership functions). The

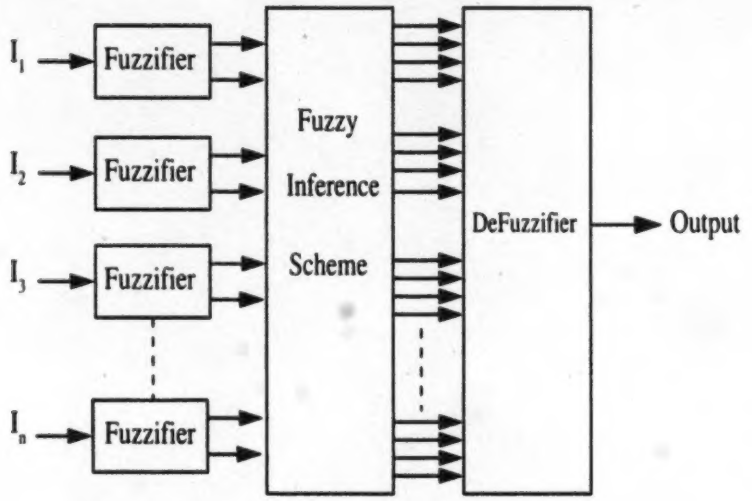


Figure 2.2: Block diagram of Fuzzy Logic Controller

crisp input is mapped to these fuzzy sets. In the Figure 2.3, any x input may be associated with one or more values of the triangular membership functions. The intersecting value of x with the fuzzy sets determines its association or fuzziness to that particular set. Since, we have assumed a degree of overlap of two; an x input can be associated with two fuzzy sets at the most.

2.3.4 Inference Scheme

The rule base determines the output on the basis of the inputs combinations and a pre-determined set of rules. A rule, in the 'if (input) ... then (output)' form is set for every possible input combination. The degree of overlap determines the number of valid rules evaluated for a single input combination. The number of valid rules evaluated is given by the Equation 2.4.

$$T_{Active\ Rules} = \{Inputs\}^{\{Degree\ of\ Overlap\}} \tag{2.4}$$

Figure 2.5 shows the mapping of the input to output membership functions. A degree of a membership is associated with every input which contributes to the rule evaluated. Since there could be more than one rule active for a given input, a method to combine the result of these active rules is essential. Two most commonly used methods of aggregation are described.

2.3.4.1 Min-Max

The 'Min-Max' algorithm combines the consequent for each rule, based on the intersection of the degree of association. Thus, the degree of membership for the

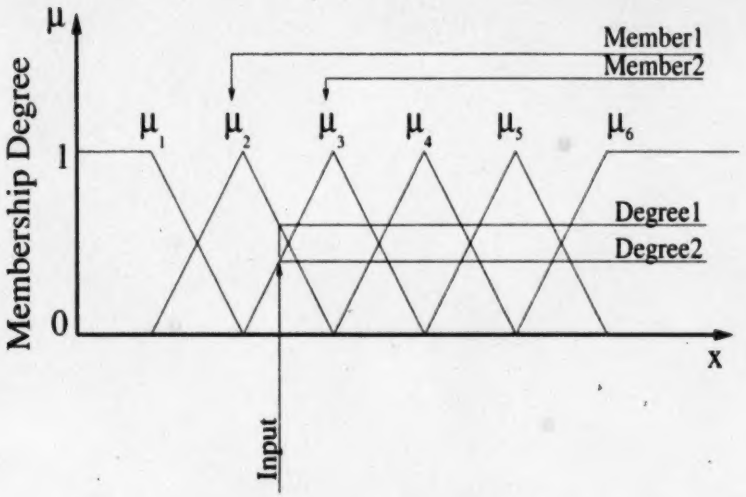


Figure 2.3: Triangular Membership functions

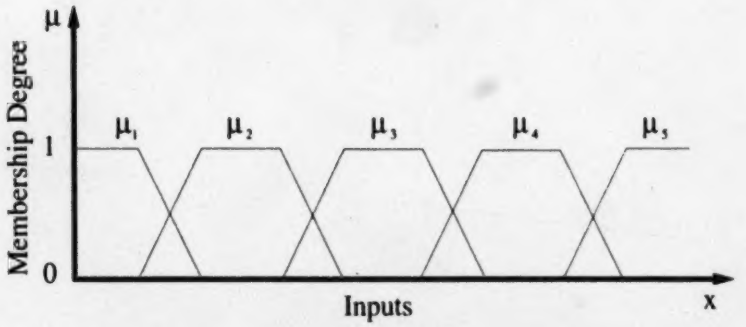


Figure 2.4: Trapezoidal Membership functions

output is the intersection (minimum) of that of the inputs. The output is the aggregate (maximum) of all the consequents. Figure 2.6 shows the logical representation of the algorithm.

Based on Fuzzy operators, 'Min' operation is carried out at the input, which is a minimum of the two inputs in Figure 2.6. At the output a 'Max' operation is performed, which combines the 'Min' operations for the two cases. Finally, the aggregate output is defuzzified to obtain the final crisp output.

2.3.4.2 Product-Sum

As the name suggests the Product-Sum algorithm uses the product operator to combine the implication and sum to aggregate the rules.

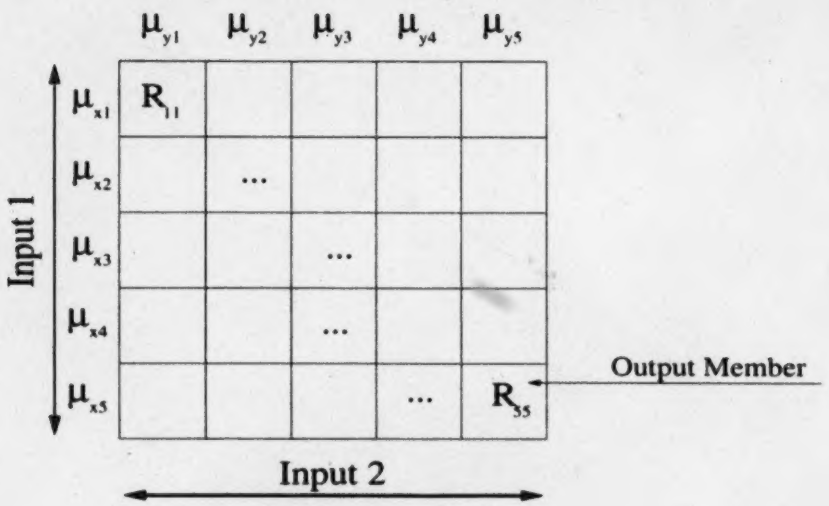


Figure 2.5: Rule Matrix

2.3.5 Defuzzifier

The defuzzifier combines the fuzzy outputs, as chosen by one or more combinations of rules to a crisp output, which forms the final stage of the system. The defuzzifier works on the results obtained from the active rules. The discussion on defuzzification is limited to the two most common algorithms.

2.3.5.1 Centre of Gravity

This type of defuzzifier implements Equation 2.5 to determine the average value of the outputs obtained from the active rules.

$$output = \frac{\sum_{i=0}^{n} \theta_i * y_i}{\sum_{i=0}^{n} \theta_i} \tag{2.5}$$

Where, θ_i is the degree of association for the fuzzified input X_i

2.3.5.2 First of Maxima Criterion

This algorithm searches for the location of the maximum value among the evaluated rules. It returns the first searched value with the maximum degree of association.

The relative positioning of the output resolved using the two methods of defuzzification is illustrated in Figure 2.7.

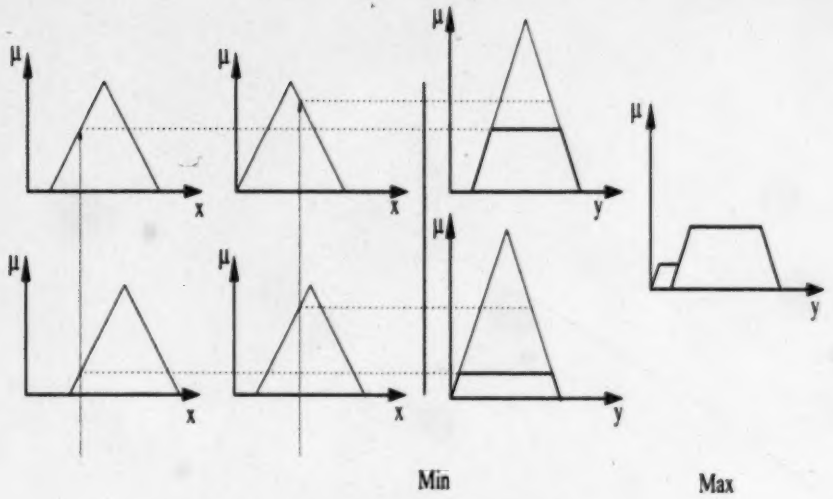


Figure 2.6: Method of Inference: Min-Max

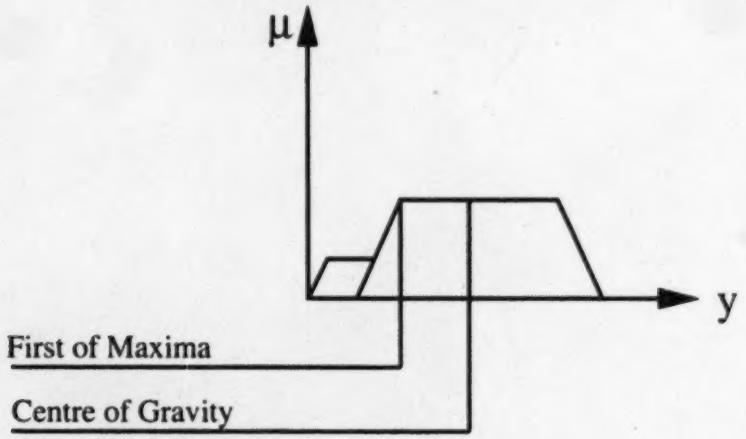


Figure 2.7: Method of Defuzzification

2.4 Review of Fuzzy Logic Controllers

The main design concern while integrating Fuzzy Logic controllers in embedded system applications is its response time in terms of the speed of inference, the area occupied and the power consumed, [31]. This section looks at some of the existing architectures for the FPGA implementation of a Fuzzy Logic controller.

The implementation of Fuzzy logic controllers have been explored for many control dominated and consumer applications to speedup the process of the inference scheme. Among the implementation techniques, they can be classified as

analog and digital controllers. In [32], the authors discuss the implementation of fuzzification and defuzzifications schemes using analog and digital hardware strategies. Trapezoidal and triangular membership functions are generated using analog and digital components. Further, two analog VLSI implementations of defuzzification circuits are presented. These circuits can be easily embedded within other systems for fuzzy implementations. Apart from using dedicated hardware, Microcontrollers and DSPs have also been used to implement FLCs. DSPs turn out to be a effective in terms of cost and speed [33]. In [34] a *reduced instruction set* (RISC) architecture processing unit is proposed, to extend the general purpose computing to achieve high fuzzy processing performance. A significantly high performance improvement is achieved with a minimal increase in chip area.

The fuzzy inference scheme has been a focus in many implementations techniques. On the basis of the inference scheme, FLCs may be broadly classified as memory-based and memory-less systems.

2.4.1 Memory based Fuzzy Logic Controller Implementations

Memory-based or off-line technique of FLC implementations stores pre-computed values of the input and output combinations. As a dedicated memory (usually external) stores all the values in the form of a look up table, there are very few limitations in terms of the flexibility of the design, [31].

In [35] a fuzzy logic controller is designed for a trigger system for High Energy Physics Experiment, which requires a fast fuzzy processor. This fuzzy processor is composed of RAM-based lookup tables to store the membership functions of the input variables and the possible rule contributions. An active rule selection block was designed to reject as many no-contribution rules as possible. The design also allows four modes of operation, allowing the design to be modified according to the application. This design supports four combinations of inputs and outputs was realized on 1.0μm CMOS technology, up to 100 *Mega Fuzzy Inference Per Second* (MFIPS). The Yager inference defuzzification method is chosen to reduce hardware implementation.

In [33], a comparison is made between the speed limits in synchronous and asynchronous design techniques. A synchronous pipelining configuration is suggested to overcome the delay and achieve an improvement in speed and reliability. The input and output membership functions are pre-computed and stored in memory to allow flexibility in deciding the type of membership function. The rule base is a combination of multiplexers feeding a memory table. The delay due to the divider at the defuzzification stage is replaced by a multiplier, the input to which is reciprocal pre-computed and store in memory. The entire FLC is implemented as a feed forward layered structure.

In [36] a generic structure of a field programmable logic device-based fuzzy logic system is described. The architecture is made flexible by storing the membership functions (input and output) and the inference rule base in dedicated memory

locations. A finite state machine or a dedicated processor acts as the main control unit. The resulting architecture operates at a frequency of 15 MHz and allows distribution of resources on hardware and software.

2.4.2 Memory-less Fuzzy Logic Controller Architectures

Memory-less or online Fuzzy Logic controllers avoid storing the pre-computed fuzzy values. Dedicated hardware resources provide online computation of the membership functions.

A CMOS 0.7μm design of a parallel, pipelined digital fuzzy processor is described in [37]. Here, a parallel pipelined architecture is proposed to allow fast selection of active fuzzy rules, which takes 20ns to execute and the chip operates at a maximum frequency of 50 MHz. Circuits generating trapezoidal membership functions are provided in the premise block and store in the memory banks. Then, the degree of association or θ is computed by another circuit block. The fuzzification unit generates the rule addresses to determine the active rules. The Sugeno zero order defuzzification circuit is pipelined to allow fast computation.

Since the use of RAM-based design is expensive in terms of area, in [38] and [39], the authors propose a dynamic rule configuration using programmable logic. Membership function generators were used to perform the input fuzzification and these antecedents were fed to a rule evaluation circuit in a programmable fashion. Finally, the centre of gravity defuzzification circuit computes the output.

In [40], the authors describe the implementation of the fuzzy logic controller on a Virtex-II FPGA using VHDL. The modular design of the FLC consists of a fuzzifier unit, an inference unit and a defuzzifier with 8-bit resolution. Only the fuzzy inference scheme is stored in an external memory. The design was pipelined to increase the data throughput of the system by avoiding any wait states. The design was tested as a Proportional Fuzzy Logic controller and was observed to operate at a maximum frequency of 21.66 MHz. In [41] an FPGA implementation of FLC is discussed where pre-computed values for the input membership functions are stored in LUTs. The major disadvantage in FPGA based architectures is identified as the limitation in resource capability. The authors overcome this drawback by introducing runtime reconfiguration and selectively configuring the FPGA with different subtasks. They also propose the idea of using multiple FPGAs to partition the design separate FPGAs.

2.5 FPGA architectures

A trend in the use of FPGA for fuzzy logic controllers can be attributed to the speed of operation and easy reconfiguration in FPGAs in place of general purpose microprocessor and microcontrollers. The advantages of using FPGAs in system was discussed earlier in Section 2.1.3. FPGAs have advanced from the rapid prototyping environment to the reconfigurable computing scenario. The new generation of

FPGAs provided soft and hard embedded cores for design applications. Following are some applications:

- Reconfigurable Computing
- Hardware Reuse
- High Speed processing
- Adaptive architectures
- Fast prototyping

Following is a brief summary of currently existing FPGA architectures from Xilinx.

2.5.1 Virtex-II Architecture

Figure 2.8 gives the architectural overview of the Virtex-II FPGA from Xilinx. The FPGA architecture comprises of *configurable logic blocks* (CLB), dedicated memory blocks, embedded multipliers and a digital clock manager.

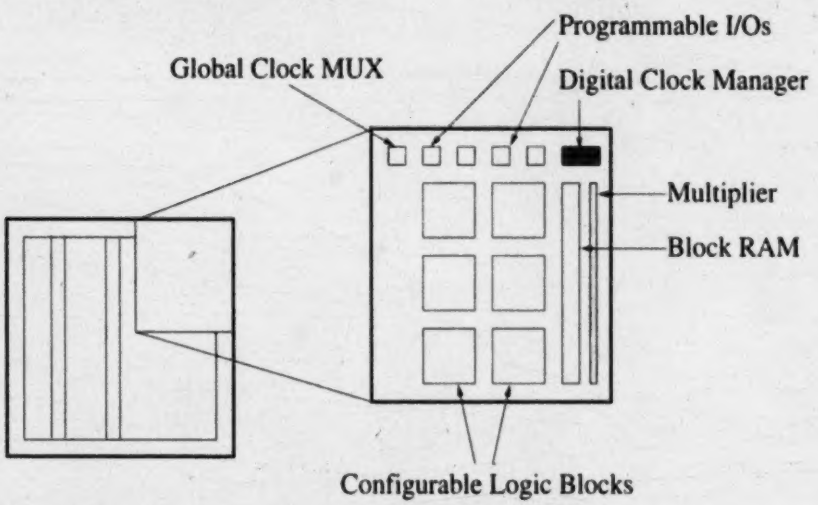


Figure 2.8: Virtex-II Architecture, [1]

2.5.1.1 Configurable Logic Blocks

The FPGA architecture comprises of a collection of configurable logic blocks interconnected through a routing matrix. Each configurable logic block is made up of four slices, as shown in figure 2.9 which constitute the distributed memory. The

distributed memory can be configured as a *look-up table* (LUT), *Random access memory* (RAM) or a *Read only memory* (ROM) or a *Shift Register* (SRL), as illustrated in figure 2.10.

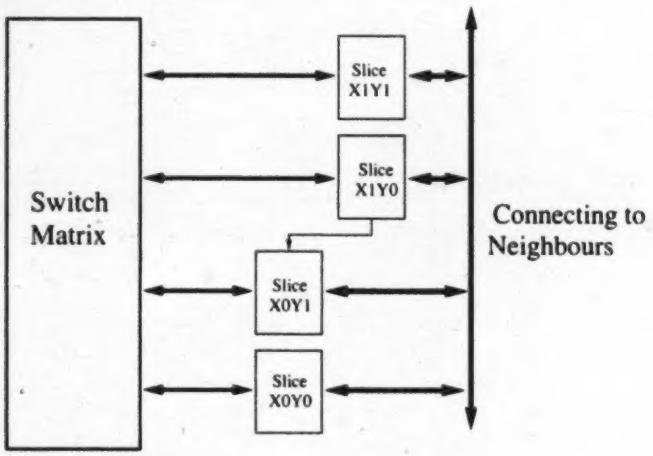


Figure 2.9: Configurable Logic Block, [1]

2.5.1.2 Embedded Block RAM

The dedicated memory blocks are 18 Kbit synchronous dual port SelectRAM blocks. The configuration of these memory blocks can be varied in terms of the address and data bus widths as the aspect ratios. They can also be configured as single port and dual port synchronous memory banks.

2.5.1.3 Multipliers

In the Virtex-II architecture has a number of 18bit X 18bit unsigned embedded multipliers. These multipliers are optimized for high speed and low power operations.

2.5.2 Virtex-II Pro

Virtex-II Pro is built on the Virtex-II architecture, but has certain additional enhanced features. It has up to two IBM PowerPC RISC processor blocks. Also, it includes up to twenty RocketIO(tm) or RocketIO X embedded *Multi-Gigabit Transceivers* (MGTs). The Figure 2.11 shows the location map of the hard-processor cores in the FPGA.

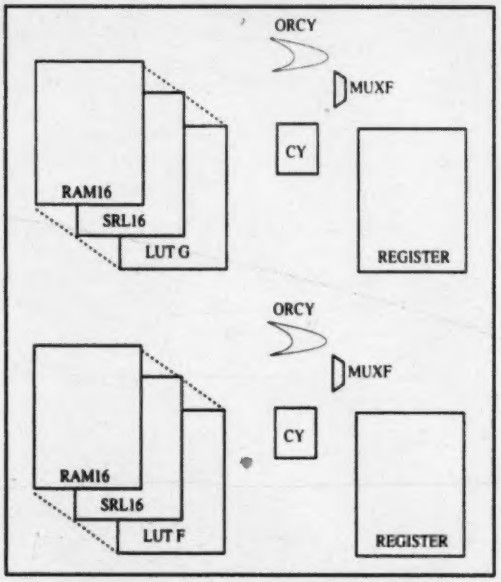


Figure 2.10: Virtex-II Slice, [1]

2.5.3 Virtex-IV

These are the newest generations of FPGAs from Xilinx. These FPGAs offer a maximum operating speed of 500 MHz with about 200,000 logic cells. The performance and density offered is twice the capabilities of other FPGA families at half the power consumption. This family comprises of 17 independent FPGAs customized to meet three specific applications. Like Virtex-II Pro, Virtex-IV also has embedded PowerPC cores suited for embedded applications. Additionally, it includes ultra high performance Digital Signal processing features with the XtremeDSP slice. These slices allow high performance at very low power.

2.5.4 Microblaze Processor Architecture

Figure 2.12 depicts the block diagram of the Microblaze soft processor. It is a reduced instruction set (RISC) embedded processor, optimized for implementation in Xilinx FPGAs. The embedded processor has thirty-two 32 bit general-purpose registers. The instruction word is 32 bit wide and has two addressing modes.

2.5.4.1 Bus Architecture

Separate 32-bit instructions and data buses exist which conform to IBM's On Chip peripheral Bus specification. Also separate 32-bit instruction and data bus exists which connects the on-chip block RAM through a *Local Memory Bus* (LMB). The

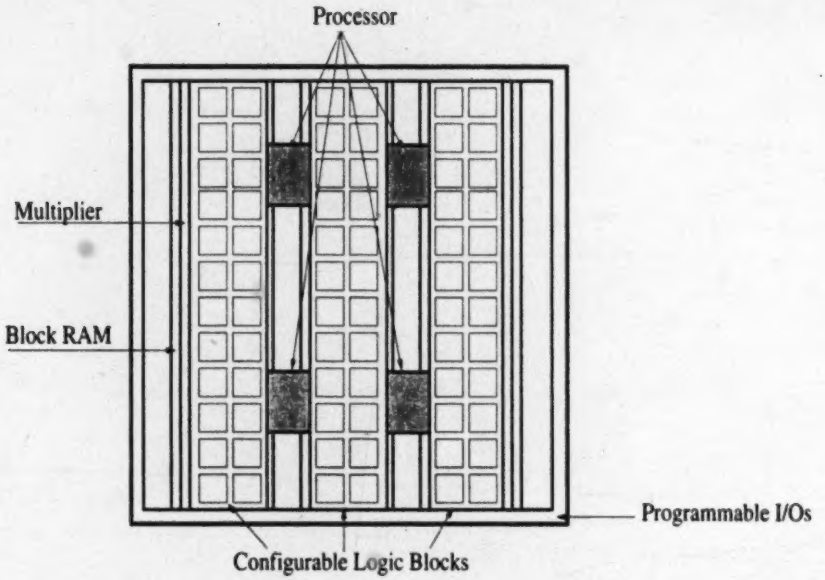


Figure 2.11: Virtex-II Pro, [2]

LMB provides single-cycle access to the on-chip dual port RAM. The *On chip peripheral bus* (OPB) provides access to both on and off chip memory and provides synchronous access for efficient block RAM transfers. It provides a guaranteed performance of 125 MHz for local memory subsystem. Apart from these buses, Microblaze also has eight-input and eight-output *Fast Simplex Links* (FSL). These links are 32-bit wide, unidirectional point-to-point data streaming interfaces for control and data transmission.

2.5.5 Cell Matrix

Cell Matrix defines a self-configurable hardware and software architecture, which implements circuits and systems, [4]. This architecture so developed is computationally complete and can be used to build any digital circuit or system. It is a multidimensional physical structure built using special cells. The hardware structure is homogeneous and is made up of these special cells, interconnected to each of its nearest neighbor. The so formed structure is capable of configuring itself, by each cell configuring its nearest neighbor.

The building block of this structure is a cell. These cells are programmable, gate level processors, each connected to its nearest neighbor. These cells perform logic functions on their inputs and forward the outputs on their 'n' sides. The function performed by the cell is dictated by the memory that holds the truth table corresponding to the n desired functions. The functionality of the cell can be changed by any of the neighbors by merely reprogramming the truth table in the configura-

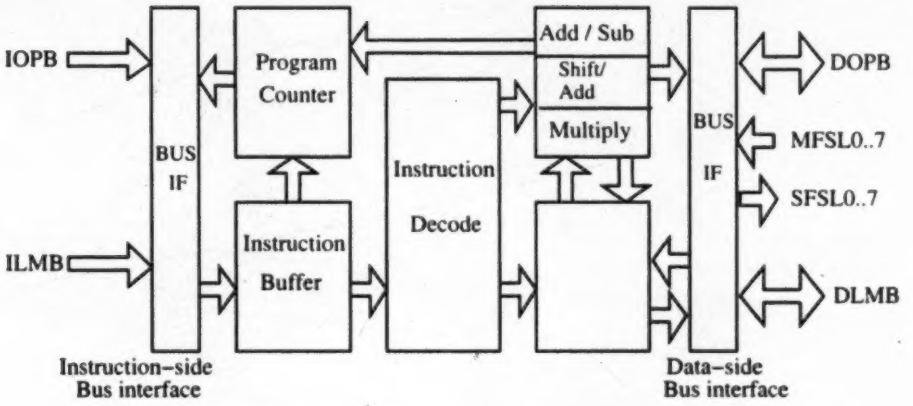


Figure 2.12: Microblaze Architecture, [3]

tion mode (C mode). It functions as a data processor in the data mode (D mode). Following are some of the structure that could be conceptualized using these cells.

This structure of the cell makes it fault tolerant, scalable, distributed and massively parallel. With these features Cell Matrix can be used to build circuits that are highly parallel circuits or self-modifying, self-assembling or self-organizing. These features make it most suitable to be used in evolvable hardware algorithms that could be cast in the Cell Matrix hardware.

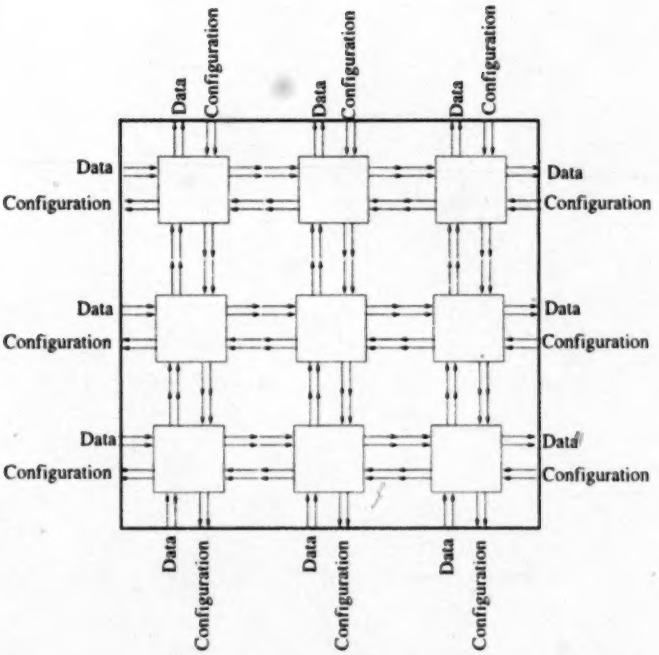


Figure 2.13: CellMatrix Architecture, [4]

Chapter 3

Evolutionary Optimization based Partitioning: Approach

This chapter gives a detailed description of the system under consideration and the problem governing the process of optimization. A section is dedicated to the architecture and the reasoning behind the choice. The remaining part of this chapter describes the procedure for solving the problem of partitioning using Genetic Algorithms.

3.1 Generic System Model

The system block diagram is shown in Figure 3.1. The architecture under consideration comprises of both preconfigured and reconfigurable hardware resources on an FPGA, software resources on GPPs and a shared memory (Multi-port RAM). The reconfigurable hardware resource allows partial reconfiguration at run-time. The software resource does not require any reconfiguration time for a change in functionality of the system resource requirement. A single large memory allows independent access to both the hardware and software resources. This is in contrast to other architecture previously considered in [18]. The main purpose of this architectural approach is to avoid bus-contention and arbitration for memory access.

Also, there exists an additional local memory which is accessible by the resources of the same type. This organization enforces the communication time between resources of the same type to be less than the communication time between the two different types of resources.

3.2 Problem Description

The application to be executed on this architecture comprises of a sequence of functionally varying tasks. With a set number of resources available, GA is used to optimize the system performance in terms of the execution time, power consumed and

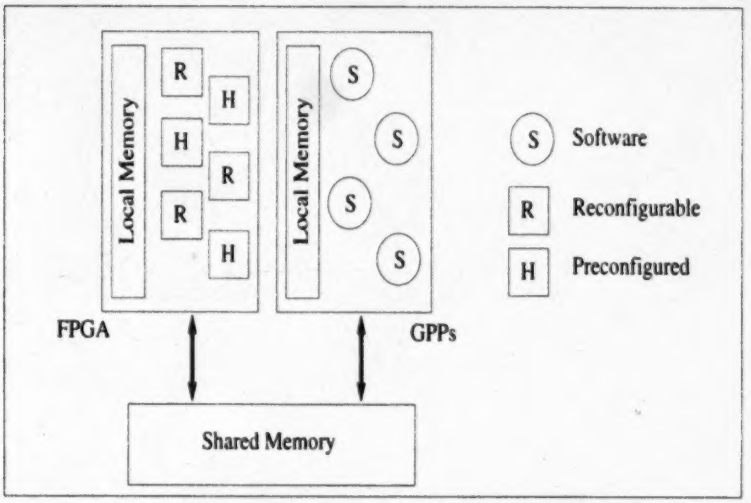


Figure 3.1: Hardware-Software System Architecture

the multi-objective optimization of time and power.

The mapping of the partitioning problem to genetic algorithms is summarized in the Figure 3.2.

3.3 Chromosomal Representation

The total number of tasks (nodes) in the task graph determines the length of the chromosome. A binary chromosome as shown in Figure 3.3 represents a system to classify tasks between hardware and software, since there are two types of possible resources. A task implemented on hardware is represented by '1' and that on software is represented by '0'. A chromosome with trinary values as shown in Figure 3.4 is used to classify tasks on hardware, reconfigurable hardware and software. A '0' represents software, '1' represents hardware with fixed configuration and a '2' represents reconfigurable hardware, classifying tasks between software, reconfigurable hardware and fixed preconfigured hardware.

With this chromosomal representation, all possible combinations that can be represented are valid. Hence, there is no further process of validation required even after any manipulation of the chromosomes in terms of crossover and mutation.

3.4 Fitness Function

Partitioning determines the type of resource allocated to each task. The quality of the partition is determined by the scheduling operation (described in Section 3.4.1). For optimization in time the total execution time determines the fitness. The total

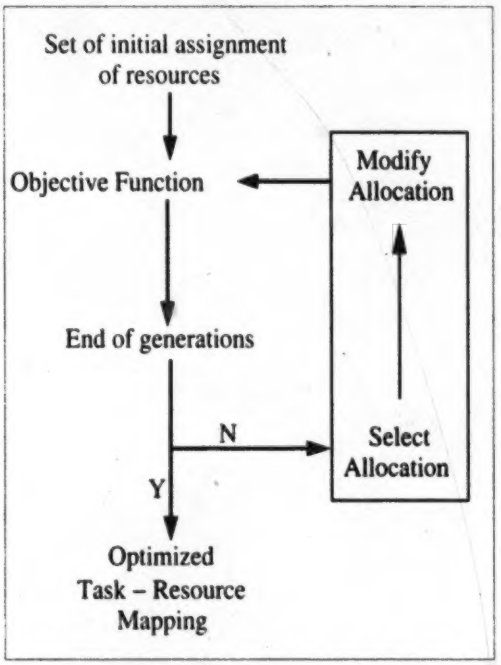


Figure 3.2: GA applied to Partitioning

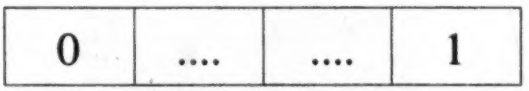


Figure 3.3: Chromosome 1

execution time is the time at which the last scheduled task completes execution. For power optimization, the total power consumed by all the resources determines the fitness. The product of the two fitness functions (time and power) is used for simultaneous power and time optimization.

With the introduction of inter-processor communication overhead, the additional time and power incurred on account of multi-processor allocation is calculated. The time overhead is on account of memory-FPGA and memory-GPP communication, which is absent in case of all-hardware and all-software design approaches. Similarly, the power overhead in the system is on account of the additional switching activity between memory and the resources.

Further, reconfiguration requires additional time and power to reconfigure the resource. A resource is reconfigured, when the configuration of the reconfigurable resource does not match that of the task to be scheduled. Once reconfigured, the actual execution time on a reconfigured hardware unit is the same as its pre-configured counterpart. Hence, it is preferred to minimize the number of reconfiguration and

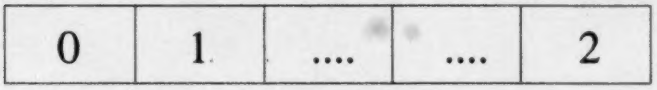


Figure 3.4: Chromosome 2

utilize the configuration for the maximum number of tasks.

In the case of resource allocation, the tasks are scheduled by varying the number of resources. In this process the number of resources allocated is forced to be the minimum possible.

3.4.1 Scheduling

A resource constrained scheduling algorithm is proposed. This is based on prioritizing tasks, explained in the following section.

3.4.1.1 Criteria for setting Task Priority

The earliest and the latest possible time instance, at which a task can be scheduled without any resource constraints, are termed as the AsapTime and the AlapTime respectively. These are set using ASAP and ALAP algorithms. Since these algorithms are applicable only to unconstrained resource scheduling, these values define the number of tasks occurring before and after a given task, in the sequence of tasks in our experiments. Initial nodes of a task graph are assumed to start at time zero. The finish time for a task is given by the sum of the start time and the execution time for that operation based on the type of resource allocated. The actual start time for each task is determined by the sum of the finish time of the preceding task and the communication delay associated between the two tasks. The task with the least (zero) number of preceding tasks is assigned the highest priority, since there are no preceding tasks that need to be performed prior to this task. The priority is based on the readiness of the task to be executed. The slack determined as the difference between AsapTime and AlapTime resolves the priority between tasks with the same start times.

3.4.1.2 Resource Constrained Scheduling

After the task priority is determined, the tasks are scheduled based on resource availability. A task is postponed when the assigned resource is in use by some preceding task. On completion of the assigned task, the resource is released for reuse by the proceeding tasks, with a lower priority. Incase of reconfigurable hardware, the execution time is decided based on its configuration and the configuration of the assigned task. In case of mismatch the resource may be reconfigured. Additional time and power is consumed to reconfigure the resource.

Hence this algorithm incorporates resource allocation along with scheduling. The task to resource mapping is defined within the same algorithm. The optimization is achieved by maximizing a function representing the design objectives, called the fitness function.

$$Maximize(Fitness) \tag{3.1}$$

The fitness function for Equation 3.1 is defined in the following sections.

3.4.2 Time Optimization

To minimize the total time taken, the fitness function f_{time}, is defined as

$$f_{time} = \frac{1}{1 + T_{norm}} \tag{3.2}$$

Where, T_{norm} is the normalized time and is given by,

$$T_{norm} = \frac{T_{total}}{T_{max}} \tag{3.3}$$

T_{total} is the time at which the last task in the priority list completes execution. T_{max} is the maximum schedule time for a given task graph. In this context, the execution time is the maximum when all the tasks are mapped to software.

T_{total} is used to assess the quality of the partition. The inherent latencies for the resource is the execution time for a task on hardware (t_h), software (t_s) and reconfigurable hardware (t_r).

Apart from the inherent latencies of the design units, additional delay is inferred in a multiprocessor environment. This has been integrated to emulate a practical system. The additional time incurred accounts for the delays due to inter-processor communication times at the edges of the task nodes.

The fitness function for time optimization, with the additional communication delay is given by,

$$f_{ctime} = \frac{1}{1 + T_{norm}} * \frac{1}{1 + C_{norm}} \tag{3.4}$$

Where, C_{norm} is the normalized communication delay and is given by,

$$C_{norm} = \frac{C_{delay}}{C_{delaymax}} \tag{3.5}$$

Where, C_{delay} is the total communication delay due to hardware-hardware (t_{hh}), hardware-software (t_{hs}), software-hardware (t_{sh}) or software-software (t_{ss}) communication overheads. $C_{delaymax}$ is the maximum communication delay. This is experienced when the delay due to communication is the most for each possible case. This is when there is hardware to software communication, or vice versa, at every operation.

3.4.3 Power Optimization

The fitness function for optimization in power, f_{power} is expressed as

$$f_{power} = \frac{1}{1 + P_{norm}} \tag{3.6}$$

Where P_{norm} is the normalized Power and is given by,

$$P_{norm} = \frac{P_{total}}{P_{max}} \tag{3.7}$$

Where, P_{total} is the sum of power consumed by all the elements in the task graph, with the given partition. P_{max} is the maximum power that can be consumed for a given task graph.

Power rating for each of the node depends on the type of resource and the type of operation. They are, power rating for hardware (P_h), software (P_s) and reconfigurable hardware (P_r). Maximum power is consumed when all the tasks are implemented on hardware (or reconfigurable hardware), since hardware resources consume higher power compared to software.

In a multiprocessor environment additional power consumed is due to the switching activity in the system on account variation in resource allocation between consecutive tasks.

For power optimization,

$$f_{spower} = \frac{1}{1 + P_{norm}} * \frac{1}{1 + S_{norm}} \tag{3.8}$$

Where, S_{norm} is the normalized switching activity and is given by,

$$S_{norm} = \frac{S_{total}}{S_{max}} \tag{3.9}$$

Where, S_{total} is the total switching power due to inter-processor communication. S_{max} is the maximum switching power experienced.

Maximum switching power is consumed when tasks alternate between hardware and software resource allocation, resulting in memory access at every node. The activities are hardware-software (P_{hs}) and software-hardware (P_{sh}) interactions, resulting in the switching activity at the memory interface bus. The hardware-hardware (P_{hh}) and software-software (P_{ss}) interaction avoids these overheads. Here, hardware represents both preconfigured and reconfigurable hardware.

3.4.4 Time and Power Optimization

To minimize both Time and Power, the fitness function for the multi-objective optimization, f_{PandT}, is defined as:

$$f_{PandT} = f_{time} * f_{power} \tag{3.10}$$

Where, f_{time} and f_{power} are given by equation (3.2) and equation (3.6) respectively.

For simultaneous optimization in power and time with the additional overheads of communication delays and switching power, the fitness function is given by,

$$f_{PandTAdd} = f_{ctime} * f_{spower} \tag{3.11}$$

Where, f_{time} and f_{power} are defined in equation (3.4) and equation (3.8) respectively.

The fitness function attempts to compete with two conflicting objectives. The time optimization attempts to minimize the total time by forcing all operations to hardware (and reconfigurable hardware), where as the power-aware system tries to avoid hardware resources. These orthogonal objectives result in hardware-software solutions, with hardware resources in time critical paths. To obtain a balance in both the objectives, the fitness function for the multi-objective optimization is the product of the two fitness functions. This results in simultaneous minimization of the two objectives.

3.4.5 Optimization for Resource Allocation

This problem of hardware software partitioning is extended to allocate optimal number of resources to proportionally allocate resources with high utilization and reduce the number of under-utilized extra resources.

Design considerations limit the numbers of resources that can be used by the given application. These limitations exist on account of area and cost constraints. Hence it is appropriate to perform task scheduling based on resource availability. However, among the resources allocated, not all are effectively utilized. GA is used to arrive at the best number of resources of each type. The architecture under consideration comprises of three types of resources, namely, preconfigured hardware, reconfigurable hardware and software. Resource allocation is effective when all the resources allocated are completely utilized.

The objective of resource allocation is to arrive at the optimal combination of resources to allow efficient utilization of system resources. The architecture under consideration has resources with varying system parameters. The utilization ratios of these resources vary with design objectives. Hence for a given design objective, GA is used to determine the type and number of resources suited to meet the design requirements. Optimal resource allocation mainly avoids resources which are unused or under-utilized by penalizing those combinations. This reduces the idle power consumption in unused resources and also the additional area occupied by resources that are not used efficiently.

0	1	2	....	....	N1	N2	N3

Figure 3.5: Chromosome for Resource Allocation

3.4.5.1 Chromosomal representation

For optimizing resource allocation additional fields tracking the number of resources of each type are appended to each chromosome, as shown in Figure 3.5. The number of types of resource determines the number of additional alleles, here N1, N2 and N3. The value of the allele varies from a minimum of one resource to the maximum number of resource available for that particular type of resource.

3.4.5.2 Penalty for Time

The fitness function adds penalty for violating a deadline. This deadline is set based on the AsapTime calculated for the task graph. A penalty comes into existence whenever the resulting schedule exceeds the time deadline.

$$P_1 = \begin{cases} 0, & \text{if } T \leq D \\ \alpha * (T - D), & \text{otherwise} \end{cases} \tag{3.12}$$

Where, T is the time taken for the execution of a task graph. D is the hard-deadline set for a task graph. α is a constant, which determines the amount of penalty applied to the violation.

3.4.5.3 Penalty for Power

The fitness function adds penalty when the maximum power consumed in the system exceeds the set requirement. This limitation is set based on the maximum power consumed by the task graph.

$$P_1 = \begin{cases} 0, & \text{if } P \leq P_{set} \\ \beta * (P - P_{set}), & \text{otherwise} \end{cases} \tag{3.13}$$

Where, P is the power evaluation for the given combination of resources. P_{set} is the set maximum power limitation of the system. β is a constant, which determines the penalty applied to avoid the violation.

3.4.5.4 Penalty for Resource Utilization

An additional penalty is added to limit the number of resources used to avoid unused resources. This directs the search process to avoid under utilization of resources.

The Penalty function is given by,

$$P_2 = \begin{cases} 1, & \text{if } T_{Unused} = 0 \\ T_{Unused}, & \text{otherwise} \end{cases} \tag{3.14}$$

Where, T_{Unused} is the total number of unused resources.
$T_{Unused} = S_{Unused} + R_{Unused} + H_{1Unused} + H_{2Unused}$.
S_{Unused} = Unused software.
R_{Unused} = Unused reconfigurable hardware.
$H_{1Unused}$ = Unused preconfigured hardware type1 (defined later in Section 4.1.1).
$H_{2Unused}$ = Unused preconfigured hardware type2.(defined later in Section 4.1.1).

The resulting fitness function which meets the required deadline with optimal resource allocation while optimizing power is given by,

$$f_{allocate} = f_{calc} * \frac{1}{1+P_1} * \frac{1}{1+P_2} \tag{3.15}$$

Where, f_{calc} represents objective as defined earlier in equation 3.4, equation 3.8 and 3.11. P_1 is defined in Equation 3.12 for Time or Equation 3.13 for Power optimization.

3.5 Details of GA parameters

3.5.1 Initial Population

The GA partitioning tool starts with random assignments for the alleles. This assignment randomly partitions the system into hardware and software. For the case of resource optimization, a random number (within the range specified) is assigned.

3.5.2 Reproduction

A single point crossover is adapted. Mutation operation, in case of partitioning, moves a task randomly from hardware to software or reconfigurable hardware, or vice versa. For resource allocation, mutation introduces a random number of resources.

The selection procedure randomly chooses between Roulette wheel and Tournament type of selection. Reproduction retains the best chromosome in each generation, using the elitist selection criteria [28].

3.5.3 Termination Criteria

GA terminates after executing a predetermined number of iterations (generations). In our experiments this number was arrived at, after running a few running a few trial runs. It is seen that, towards the end of these fixed generations the fitness function has almost stabilized.

Chapter 4

Evolutionary Optimization based Partitioning: Results

In this chapter, the first section describes the set up for the experiments. The next section provides a detailed description of the experiments performed to demonstrate the performance of the optimization procedure adapted for hardware software partitioning. A section on the parameters assumed for the set of experiments is listed. This is followed by a detailed discussion on the results.

4.1 Benchmarking Scenario

The absence of well-defined benchmarks in Hardware Software co-design restricts the true performance comparison of the implementation techniques and architectures. On this account we have chosen randomly generated task graphs to represent the sequence of events in embedded system architecture. Future work may be compared to these results, by regenerating the same task graphs based on the parameters listed in Appendix A. These task graphs were generated using *Task Graphs For Free* (TGFF), described in [15]. They provide the sequence of operations in a system. This in turn provides the precedence relationship between tasks. A sample task graph is illustrated in Figure 4.1.

4.1.1 Task Graphs

A set of five task graphs with a node count of 200 (on an average, with a variance of one) was generated. The nodes in the task graph were restricted to two types, based on the number of inputs to each node. The variation in the number of input to the nodes is mapped as a variation in the task functionality. A two input node is considered to be functionally different from a one input node (for e.g. two input node corresponds to an adder and a one input node corresponds to a multiplier). Attributes such as execution time, cost, area and power may also be added to each of the tasks to be randomly assigned.

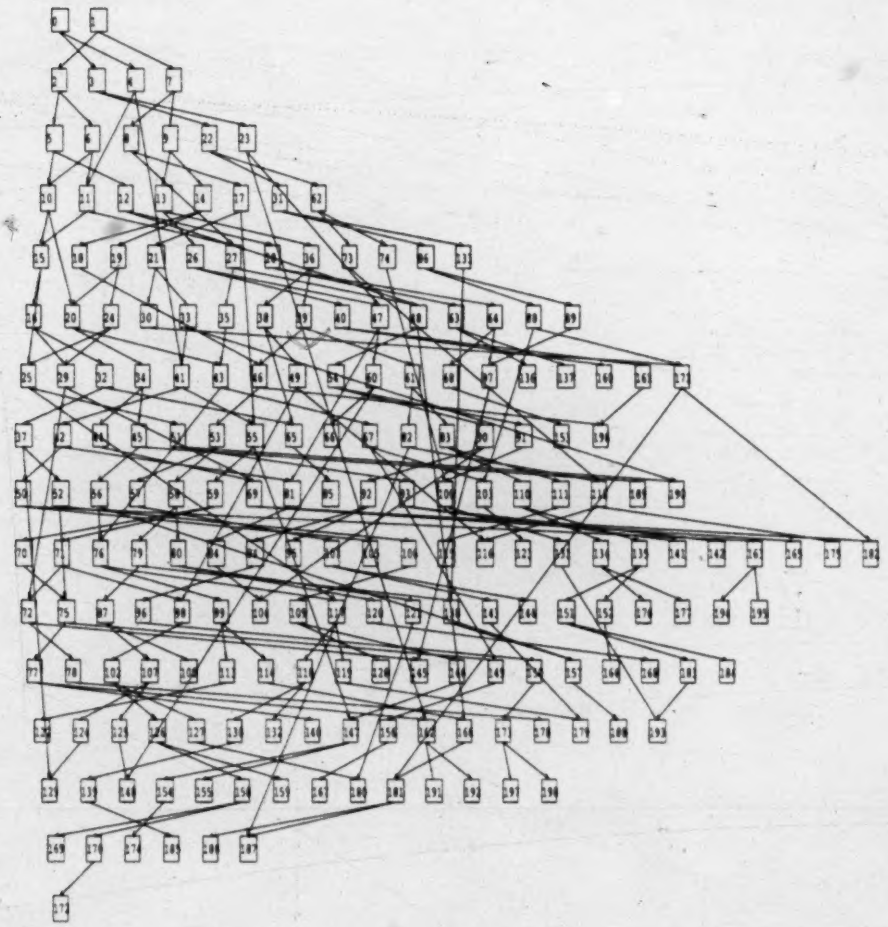


Figure 4.1: Task Graph with 200 nodes

System resources have predefined execution time and power rating for resources. Individual task deadlines are ignored. To accommodate two types of tasks, two types of resources are required (type1 and type2). Consequently, on the variation in the functionality the two nodes differ in terms of the execution time and power consumed. Reconfiguration allows change in functionality between the two types, at the cost of an additional overhead of the reconfiguration time and power.

4.1.2 Design Parameters

The design parameters entirely determine the system architecture. Simple variations in the timing parameters can change the architecture of the system from a uniform shared memory architecture to a non-uniformed shared memory architecture, (illustrated in Appendix C). For our experiments we consider shared memory architecture.

These assumptions are made only for illustrative purposes, in order to logically evaluate the algorithm used. These parameters may be changed according to the actual architecture of a real application. The comparative relations between can be expressed as follows:

$$t_s = 10 * t_h \tag{4.1}$$

$$t_h = t_r \tag{4.2}$$

Where, t_s is the execution time of a task on a software resource and t_h is the execution time of the same task on a preconfigured hardware resource. Similarly, t_r is the execution time on a reconfigured hardware resource.

The inter-processor communication is expressed as:

$$t_{sh} = t_{hs} \tag{4.3}$$

Where, $t_{sh} = t_{sm} + t_{hm}$, is the communication delay between the software to hardware via the shared memory. t_{sm} and t_{mh} are the software access time to memory and hardware access time from memory respectively. This assumption leads to the following relations,

$$t_{hh} < t_{sh} \tag{4.4}$$

$$t_{hh} = t_{ss} \tag{4.5}$$

Where, t_{ss} is the communication time between two consecutive tasks on software resources. Similarly, t_{hh} is the communication time between to consecutive hardware tasks.

Similarly for power,

$$P_s < P_h \tag{4.6}$$

Where, P_s is the inherent power consumed by a task when implemented on a software resource and P_h when implemented on a hardware resource.

The additional power consumed can be expressed as follows:

$$P_{sh} = P_{hs} \tag{4.7}$$

$$P_{hh} > P_{ss} \tag{4.8}$$

Considering the activity during inter-processor communication, leads to the assumption that P_{sh} and P_{hs} are higher than P_{ss} and P_{hh} respectively. Where, P_{sh} is the power consumed when communicating between software to hardware resources (and P_{hs} viceversa).

These parameters were assumed in accordance with [16].

4.2 GA Parameters

After performing a series of experiments the GA parameters were fixed. A population size of 4000 was set for all the experiments. The initial population was created randomly. GA was set to evolve over 200 generations. The reproduction parameters were set as $P_c = 0.9$ and $P_m = 0.005$. The time and power parameters assumed for the experiments is listed in Table B.1 and Table B.2 in Appendix B.

4.3 Experiments

4.3.1 Experiment 1

To validate the assumptions made in Section 4.1.2 and to confirm the consistency, the proposed algorithm was executed multiple times (10), with each run initialized with a randomly generated seed. Further, a comparative study was conducted to observe the variations of the results obtained with the introduction of runtime reconfiguration.

4.3.1.1 Case 1: Software and Hardware (Only Preconfigured)

The system under consideration comprises of two types of resources, they are software and hardware. The hardware resources are assumed to be fixed and preconfigured. To accommodate the two functionally diverse nodes, the FPGA has two resources of each type, pre-configured. An equal number of software resources are assumed to exist in the system. Thus, for this configuration the architecture comprises of one FPGA with four preconfigured resources and four GPPs.

4.3.1.2 Case 2: Software and Hardware (Preconfigured and Reconfigurable)

Run-time reconfiguration of hardware resource is introduced in this architecture. Consequently, to maintain the same ratio of resources, there are four hardware resources available on the FPGA. Two of these resources have fixed configurations (preconfigured) and the other two can be reconfigured at runtime. The same numbers of software resources are retained, as in Case 1. This system architecture comprises of an FPGA and four GPPs, as in Case 1.

Case 1 and Case 2 are compared for the design objectives of time, power and simultaneous optimization of time and power.

4.3.2 Experiment 2

Along with the design objective of optimization in time and power, GA is used to determine the optimal number of resources required for the implementation of task graph. Three kinds of resources are assumed to exist concurrently in the system architecture, namely, software resources, fixed preconfigured hardware and runtime reconfigurable hardware resources.

4.3.2.1 Case 1 - Time Constraint

A hard-deadline is set, to introduce a time constraint. The penalty function is applicable whenever there is a time violation. This was experimented for optimization in power and the multi-objective optimization of time and power.

To determine the hard deadline, a parameter called $H_{Critical}$ was introduced. $H_{Critical}$ was set on the basis of the calculation of the AsapTime (described in Section 3.4.1.1), with only hardware resources. Based on this value, the algorithm was executed with a hard-deadline of $H_{Critical}$, 1.5 * $H_{Critical}$ and a deadline of $2*H_{Critical}$.

4.3.2.2 Case 2 - Power Constraint

A restriction in power is introduced, with a power constraint. Similar to the hard-deadline in Case 1, a penalty function was introduced to limit the maximum system power. This case of power constraining was experimented for the optimization objective of time and the multi-objective optimization of time and power.

The maximum power limitation was determined based on the maximum power consumed by the system, defined by P_{max} and S_{max} in Equation 3.7 and Equation 3.9. The experiments were repeated for a power limitation of $(P_{max} + S_{max})/3$, $(P_{max} + S_{max})/4$ and $(P_{max} + S_{max})/6$.

4.4 Results

4.4.1 Random Search and GA

The plot in Figure 4.2 compares the execution times and power consumed for varying values of P_c and P_m. The values to the extreme left, $P_c = 0$ and $P_m = 1$) represent a case of Random search. This, when compared to the values appearing at the center of the graph ($P_c = 0.9$ and $P_m = 0.01$) and towards the extreme right ($P_c = 0.9$ and $P_m = 0.005$), show a very large variation. This illustrates the significance of the optimization procedure adapted.

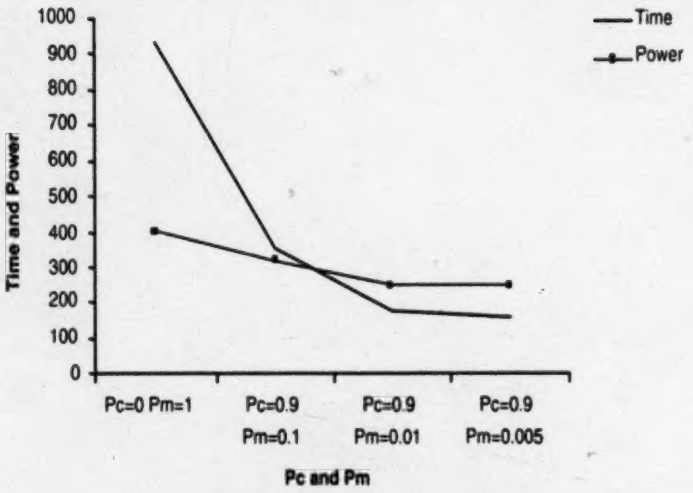


Figure 4.2: Results of GA v/s Random Search

Table 4.1: Results for Experiment-1 Case 1: Time

Task Graphs	Time	Power
Task200-5-1	**434.30 +/- 1.70**	728.70 +/- 2.49
Task200-5-2	**422.60 +/- 2.75**	734.90 +/- 3.14
Task200-5-3	**427.50 +/- 1.71**	718.70 +/- 3.97
Task200-5-4	**436.70 +/- 3.26**	721.80 +/- 2.69
Task200-5-5	**411.90 +/- 8.10**	745.50 +/- 8.92

The plot of the fitness function for the maximization objective determined for one of the experiments is shown in Figure 4.3. The plot shows the best and the average obtained over a range of generations. The fitness function stabilizes towards the end of the predefined number of generations (200 in this case), indicating optimal results. The increase in the average value also indicates that the entire population gradually improves towards the best solution.

4.4.2 Results from Experiment 1

Table 4.1, Table 4.2 and Table 4.3 summarize the performance for the optimization in time and power individually and the combined optimization of the two objectives. These results are for the resource constraints described in Section 4.3.1.1.

Table 4.4, Table 4.5 and Table 4.6 shows the performance variation with the introduction of run-time reconfiguration on hardware. The resource constraints are

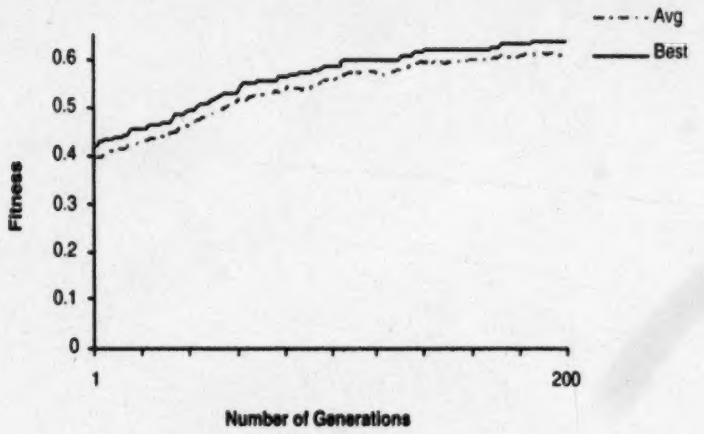


Figure 4.3: Variation in the fitness function over generations

Table 4.2: Results for Experiment-1 Case 1: Power

TaskGraphs	Time	Power
Task200-5-1	1677.20 +/- 14.52	**229.20 +/- 2.97**
Task200-5-2	1662.00 +/- 13.37	**250.20 +/- 1.75**
Task200-5-3	1653.20 +/- 18.16	**246.30 +/- 3.19**
Task200-5-4	2154.50 +/- 5.05	**245.50 +/- 1.64**
Task200-5-5	1654.90 +/- 14.53	**249.20 +/- 2.97**

as mentioned in Section 4.3.1.2.

Following sections describe the inferences made for the results obtained.

4.4.2.1 Consistency of GA

The results in Table 4.1 (and Tables 4.2, 4.3, 4.3, 4.4, 4.5, 4.6) are tabulated in { mean +/- standard deviation }. It is seen that GA arrives at the similar set of results for all the ten random executions of the algorithm. This validates the fitness function employed. Also, on close observation it is seen that the variation is less than 1% of the mean, which demonstrates that the algorithm employed is consistent at arriving at the optimal values repeatedly.

4.4.2.2 Time Optimization

For time critical applications it is seen that run-time reconfiguration enhances system performance slightly. This improvement is seen even in the presence of the additional reconfiguration overhead. The results for timing optimization so ob-

Table 4.3: Results for Experiment-1 Case 1: Time & Power

TaskGraphs	Time	Power
Task200-5-1	**436.70 +/- 1.70**	**700.50 +/- 5.75**
Task200-5-2	**427.70 +/- 4.16**	**703.80 +/- 5.94**
Task200-5-3	**432.60 +/- 5.46**	**683.00 +/- 9.64**
Task200-5-4	**445.70 +/- 8.17**	**680.10 +/- 12.23**
Task200-5-5	**445.70 +/- 7.77**	**709.40 +/- 6.51**

Table 4.4: Results for Experiment-1 Case 2: Time

TaskGraphs	Time	Power
Task200-5-1	**420.90 +/- 2.96**	1496.10 +/- 9.39
Task200-5-2	**418.10 +/- 3.54**	1509.50 +/- 6.09
Task200-5-3	**413.30 +/- 2.49**	1212.30 +/- 9.70
Task200-5-4	**418.60 +/- 3.80**	1219.00 +/- 6.28
Task200-5-5	**429.00 +/- 4.29**	1301.40 +/- 14.69

tained with reconfigurable hardware resources are better than the results with pre-configured hardware. Fig 4.4 shows the variation in time in the two cases, for all the design objectives.

4.4.2.3 Power Optimization

For power optimization, the procedure almost completely avoids hardware blocks to reduce the power consumed by the system in both the cases. The task execution relies entirely on the software resource and all the operations are queued to this resource. Hence, for power critical applications, the choice of reconfiguration may be avoided, since its introduction only deteriorates power. Figure 4.5 compares the results obtained for power in the two cases, for all the design objectives.

Table 4.5: Results for Experiment-1 Case 2: Power

TaskGraphs	Time	Power
Task200-5-1	1654.80 +/- 22.92	**464.10 +/- 7.04**
Task200-5-2	1638.90 +/- 19.72	**463.80 +/- 9.17**
Task200-5-3	1616.50 +/- 21.26	**553.90 +/- 11.34**
Task200-5-4	2101.00 +/- 28.99	**557.44 +/- 10.08**
Task200-5-5	1627.70 +/- 16.36	**471.50 +/- 7.27**

Table 4.6: Results for Experiment-1 Case 2: Time & Power

TaskGraphs	Time	Power
Task200-5-1	**436.80 +/- 9.49**	**1093.90 +/- 8.06**
Task200-5-2	**430.90 +/- 12.14**	**1098.10 +/- 13.88**
Task200-5-3	**426.40 +/- 11.98**	**1088.90 +/- 9.25**
Task200-5-4	**464.20 +/- 12.28**	**1059.10 +/- 9.52**
Task200-5-5	**438.60 +/- 9.19**	**1117.40 +/- 9.38**

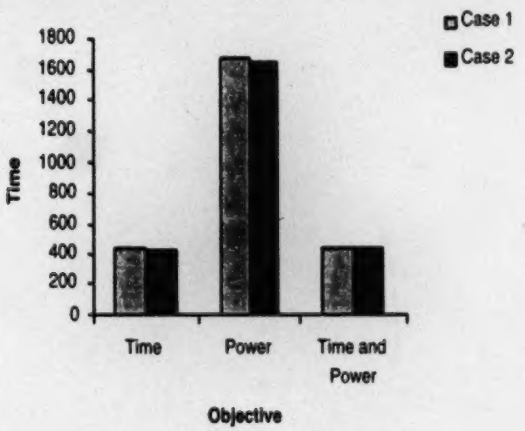


Figure 4.4: Optimization for Time

4.4.2.4 Time and Power Optimization

For the multi-objective optimization in power and time to reduce the total time and power consumed simultaneously, both hardware and software design units are utilized. From Figure 4.4 and 4.5 it is observed that, introduction of time as parameter along with power changes achieves results similar to the individual optimization of time.

4.4.2.5 Task Distribution

Figure 4.6 depicts the variation in the resource utilization in hardware and software elements, depending on the objective applied. As can be observed, an all-hardware implementation is best suited for time optimization. The software units utilized in this process merely provide an additional resource for parallel operation. Similarly, for power optimization, the procedure almost completely avoids hardware blocks to reduce the power consumed by the system. The result so obtained is the best combination for a power-optimal solution, but unacceptable for time critical

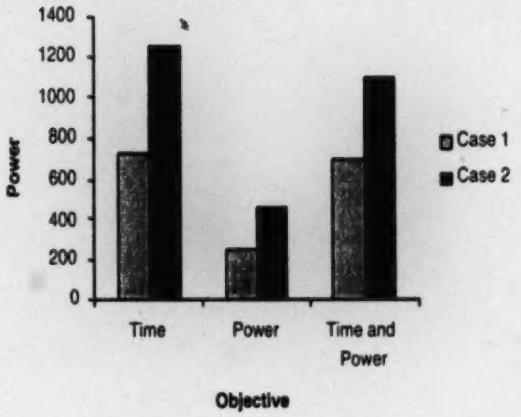


Figure 4.5: Optimization for Power

applications.

For the multi-objective optimization in power and time, the influence of inter-processor communication is insignificant. To reduce the total time and power consumed both hardware and software design units are utilized. This results in a solution better than pure-software or pure-hardware implementation.

A similar observation may be made for the reconfigurable hardware architecture, shown in Figure 4.7. Among the hardware resources utilized, the reconfigurable resources outnumber preconfigured hardware.

Figure 4.8 and Figure 4.9 summarize the variation in performance objective as seen on the distribution of tasks and resources. The timing optimization objectives individually illustrate the influence of communication delay. Similarly, the power optimization results are expressed separately for inherent power and the influence of switching power.

4.4.3 Results from Experiment 2

4.4.3.1 Results with Hard Deadline

Table 4.7 summarizes the results obtained for the optimization of power, while meeting a hard-deadline. Table 4.8 lists the results obtained for the optimization of time and power for a set hard-deadline. Limiting the execution time, implements a power efficient design well within the specified deadline. This is implemented with the minimum number of resources.

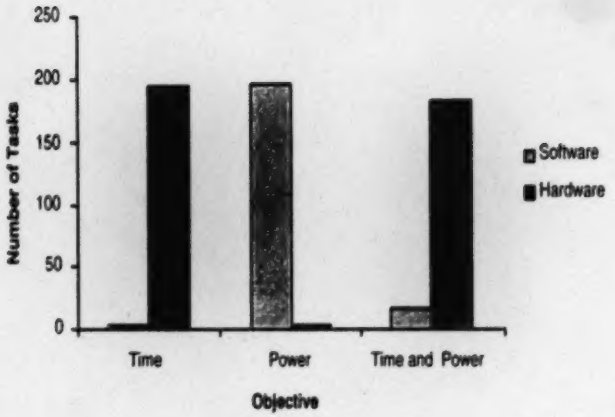


Figure 4.6: Task distribution of tasks on Hardware / Software - Case 1

Table 4.7: Power Optimization with Hard Deadline

	AsapTime	AsapTime * 1.5	AsapTime * 2
Deadline	260	390	520
Time	295	390	515
Power	**1183**	**1105**	**1031**
Software Resources	4	4	4
Reconfigurable Resources	4	3	2
Hardware Resources of Type1	4	4	4
Hardware Resources of Type2	4	3	3
Tasks on Software	8	18	32
Tasks on Reconfigurable Hardware	60	47	46
Tasks on Hardware	134	137	124

4.4.3.2 Results with Power Limitation

Table 4.9 summarizes the results obtained for the optimization of time, while meeting the maximum power constraint. Table 4.10 lists the results obtained for the optimization of time and power for a set maximum power requirement. The variation in resource allocation and task mapping causes a variation in time and power. The penalty introduced to restrict the power consumption, limits the search space. Hence, this results in timing optimization while meeting the maximum power requirement.

From these results it is seen that, as the number of tasks mapped on a resource type increases, the resource count is proportionally kept high. Similarly for low resource utilization, the resource count is kept to the minimum required. This indi-

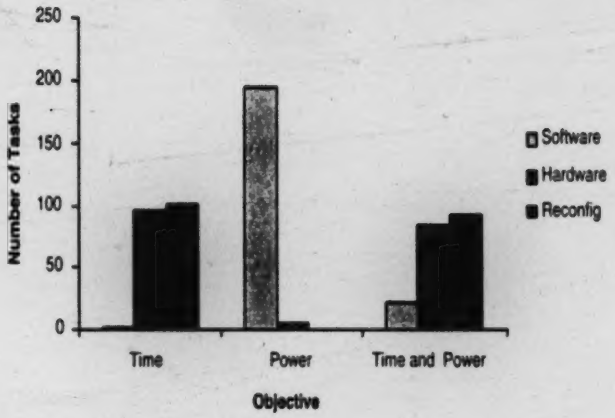


Figure 4.7: Task distribution of tasks on Hardware / Software - Case 2

Table 4.8: Simultaneous Time and Power Optimization with Hard Deadline

	AsapTime	AsapTime * 1.5	AsapTime * 2
Deadline	260	390	520
Time	**294**	**319**	**449**
Power	**1194**	**1145**	**1082**
Software Resources	4	4	4
Reconfigurable Resources	2	2	1
Hardware Resources of Type1	4	4	4
Hardware Resources of Type2	3	4	3
Tasks on Software	6	12	27
Tasks on Reconfigurable Hardware	60	43	41
Tasks on Hardware	136	147	134

cates usage of software elements in time critical applications only for tasks which do not affect the total execution time. Similarly, resource count on hardware is restricted to the minimum possible for power optimal solutions. The multi-objective optimization of power and time arrives at a distributed resource allocation.

4.5 Observation and Inference

A consistency in the quality of the results obtained using GA is observed over a wide range of design examples. The analysis in terms of the mean and variation ranges establishes the dependability of GA in the problem addressed. Retaining the same scheduling algorithm, the fitness function can be modified to redesign the

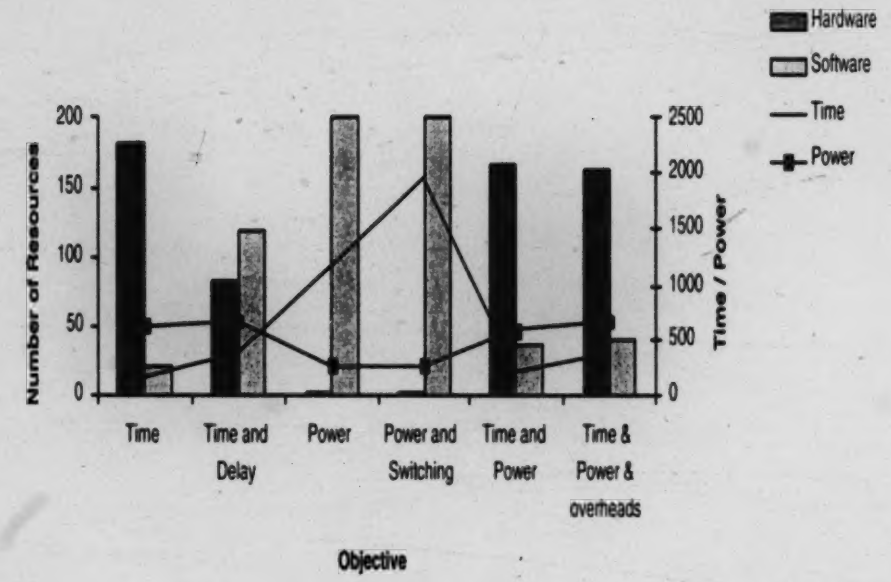


Figure 4.8: Distribution of Hardware / Software utilization in system implementation

system objective. Thus, the proposed optimization technique for time and power can be extended to other design parameters such as area and cost of silicon. The suitability of using objective-specific resources is clearly observed in the resource utilization ratios.

Resource allocation using GA provides information to build the most suited target architecture for a system application. With this approach, the number of unused resources and under-utilized devices are reduced to a minimum possible. Hence, for a given application, with set design objectives, GAs arrive at the most suited resource allocation and scheduling for the system. In short, GA evolves the design of the system. Additionally, with the introduction of penalties for time and power, the power optimization keeps a limit on the execution time. Also, the time optimization maintains power below the maximum threshold.

To introduce weights for the objective functions with a greater priority, the fitness function may be transformed as follows:

$$Fitness = f_{time} * f_{power} \tag{4.9}$$

$$Fitness = f_{time}^{\alpha} * f_{power}^{\beta} \tag{4.10}$$

Where α and β represent the weights which prioritize the objective. When α is

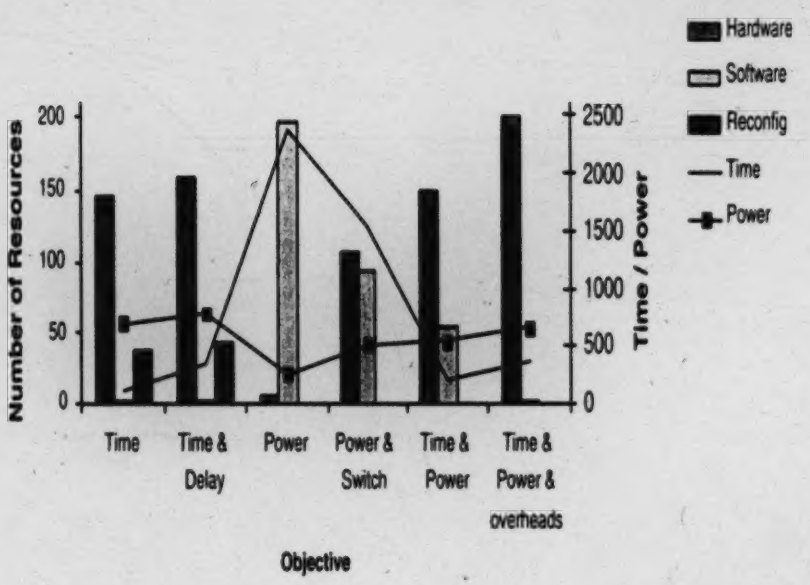


Figure 4.9: Distribution of Hardware (Preconfigured and Reconfigurable) / Software in system implementation

≥ 1, the value of f_{time} has a greater significance than with $\alpha < 1$. Similarly for β, applicable to f_{power}.

Table 4.9: Time Optimization with Power Limitation

	$(P_{max}+S_{max})/6$	$(P_{max}+S_{max})/4$	$(P_{max}+S_{max})/3$
MaxPower	400	600	801
Power	467	600	793
Time	**1678**	**1404**	**1004**
Software Resources	4	4	4
Reconfigurable Resources	1	1	1
Hardware Resources of Type1	1	1	3
Hardware Resources of Type2	1	1	1
Tasks on Software	194	156	99
Tasks on Reconfigurable Hardware	2	16	29
Tasks on Hardware	6	30	74

Table 4.10: Simultaneous Time and Power Optimization with Power Limitation

	$(P_{max}+S_{max})/6$	$(P_{max}+S_{max})/4$	$(P_{max}+S_{max})/3$
MaxPower	400	600	801
Power	**457**	**503**	**537**
Time	**1703**	**1577**	**1523**
Software Resources	4	4	4
Reconfigurable Resources	1	1	1
Hardware Resources of Type1	1	2	1
Hardware Resources of Type2	1	1	1
Tasks on Software	199	179	172
Tasks on Reconfigurable Hardware	1	4	6
Tasks on Hardware	2	19	24

Chapter 5

Fuzzy Logic Controller: FPGA Implementation

In this chapter, the design approach to implementing a Fuzzy Logic Controller on an FPGA is described. The first section details the approach to implementing the FLC entirely on the FPGA (Hardware Architecture). Strategies of designing a FLC are proposed in the same section. The second section details the option of using an embedded processor on the FPGA (Software Architecture), in contrast to the conventional approach. Finally, a section on the feasibility analysis puts forth the architectural limitation in the current implementation using the hardware architectural approach.

5.1 Hardware Architecture

The conventional *Hardware Description Language* (HDL) implementation methodology is used to design and implement a FLC on the FPGA. The design modifications incorporated to work around certain hardware issues are illustrated in this section.

5.1.1 Fuzzification

This thesis limits the discussion to only triangular and trapezoidal membership functions, as they can be constructed with very little resources. Figure 5.1 and Figure 5.2 illustrate the structure of the membership functions and the characteristic key-points. These key points are the only values to be stored in registers to perform the fuzzification, hence amount to very little resources. The process of fuzzification was introduced earlier in Section 2.3.3.

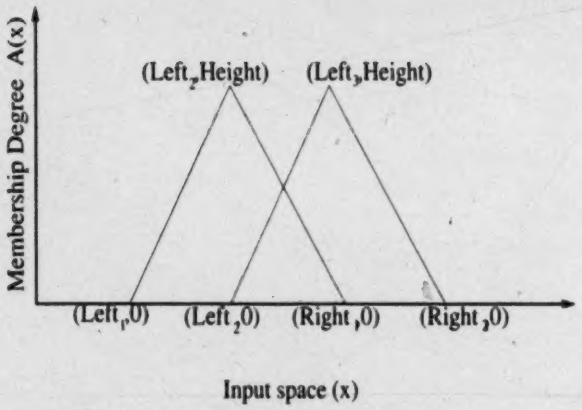


Figure 5.1: Triangular Membership functions

5.1.1.1 Triangular Membership function

The triangular membership functions are represented in terms of the co-ordinates of the base, the co-ordinates of the peak and the slope. For e.g., in Figure 5.1, the points are $(Left_1, 0), (Right_1, 0)$ and $(Left_2, Height)$, with a slope of $Slope_1$ (or $-Slope_1$). For any input (say I_1), the fuzzified value A(x) is determined using Equation 5.1, for triangular membership functions.

$$A(x) = \begin{cases} Height * \frac{I_1 - Left_1}{Left_2 - Left_1}, & if\ Left_1 \leq I_1 \leq Left_2 \\ Height * \frac{I_1 - Right_1}{Left_2 - Right_1}, & if\ Left_2 \leq I_1 \leq Right_1 \\ 0, & otherwise \end{cases} \tag{5.1}$$

5.1.1.2 Trapezoidal Membership function

The fuzzy computation for a triangular membership function can be extrapolated to a trapezoidal membership function represented in Equation 5.2. Or vice versa, Equation 5.2 for a trapezoid, reduces to Equation 5.1 when $TRight = TLeft$.

$$A(x) = \begin{cases} Height \frac{I_1 - Left_1}{TLeft_1 - Left_1}, & if\ Left_1 \leq I_1 \leq TLeft_1 \\ Height, & if\ TLeft_1 \leq I_1 \leq TRight_1 \\ Height \frac{I_1 - Right_1}{TRight_1 - Right_1}, & if\ TRight_1 \leq I_1 \leq Right_1 \\ 0, & otherwise \end{cases} \tag{5.2}$$

5.1.1.3 Assumptions to simplify FPGA implementation

- All the numbers (Fuzzy and Crisp) have an 8-bit representation. The values range from 0 to 2^8-1.

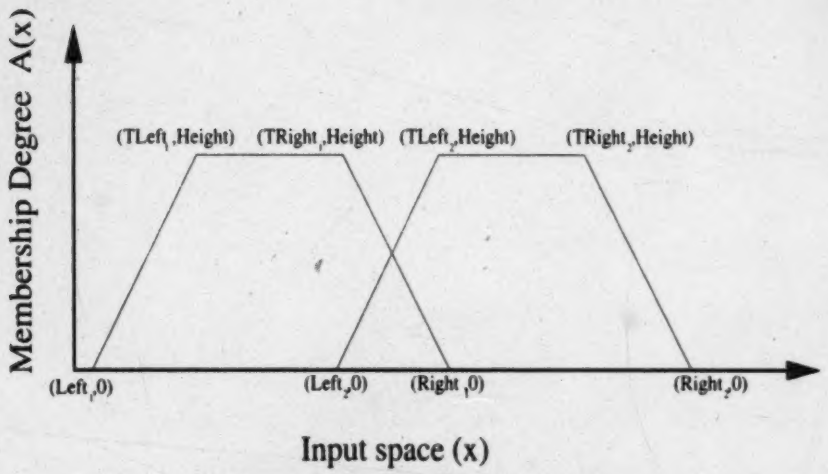


Figure 5.2: Trapezoidal Membership functions

- A(x) also defines the degree of membership. This assumption avoids all floating (or fixed) point arithmetic units.
- The degree of membership is represented as an integer ranging from 0 to 2^8-1, instead of a floating point value with a range from 0 to 1.

5.1.1.4 Resources required

- Comparators: To determine the overlapping membership functions and to locate the point of intersection.
- Multipliers and Subtractors: To perform Equation 5.1(or Equation 5.2) for each intersecting membership functions.

Figure 5.3 shows the block level representation of the Fuzzifier. A detailed schematic which includes the operators inferred is shown in Figure 5.4. The type and number of hardware resources required for the fuzzification may be approximated with the following relations.

$$T_{Mult} \propto O_{deg} * Inputs \tag{5.3}$$

$$T_{Subtr} \propto O_{deg} * Inputs \tag{5.4}$$

$$T_{Comp} \propto M_{bits} \tag{5.5}$$

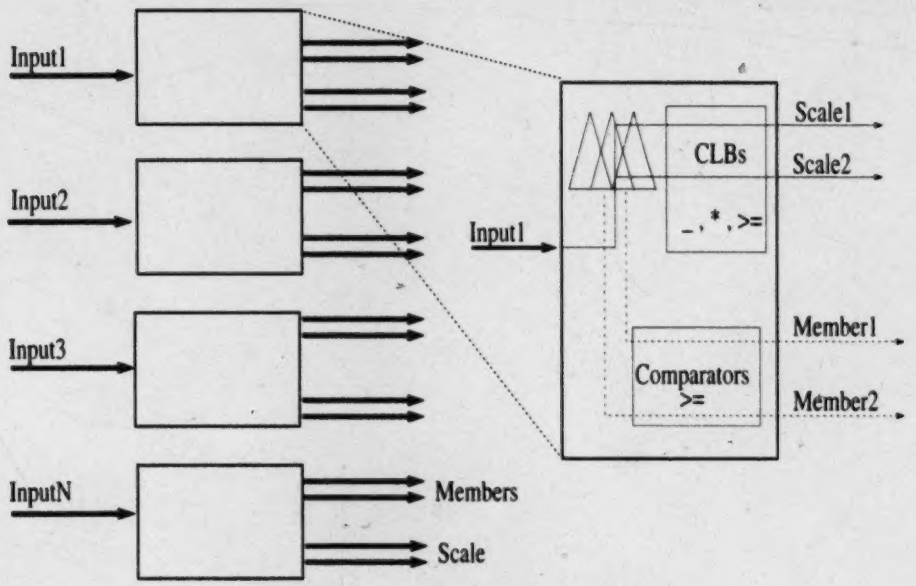


Figure 5.3: Schematic for Fuzzification

Where, T_{Mult}, T_{Subtr} and T_{Comp} are the total number of multipliers, subtractors and comparators respectively. O_{deg} is the degree of overlap and M_{bits} is the number of bits required to represent the membership functions.

5.1.2 Rule Evaluation

The inference scheme has two possible implementation options. The combinatorial implementation maps the rules onto the CLBs, which results in a large amount of area occupied. Alternately, the rules can be mapped to the embedded Block RAMs in the FPGA. This option frees the CLBs and can be used to implement other resources.

5.1.2.1 Combinatorial Implementation

The output is defined for all possible combinations of the input fuzzy sets. Since the output is made up of fixed number fuzzy sets, the inputs to the system act as inputs to multiplexers choosing one of the possible outputs. Hence, CLBs are utilized to combine the inputs and infer a logic structure (multiplier), which defines the output. Figure 5.5 shows the internal schematic for the combinatorial implementation.

Consequently, the CLB resource utilization depends on the number of rules defined. The number of rules defined depends on the inputs to the system and the number of fuzzy sets associated with each input. The relationship is defined as in Equation 5.6

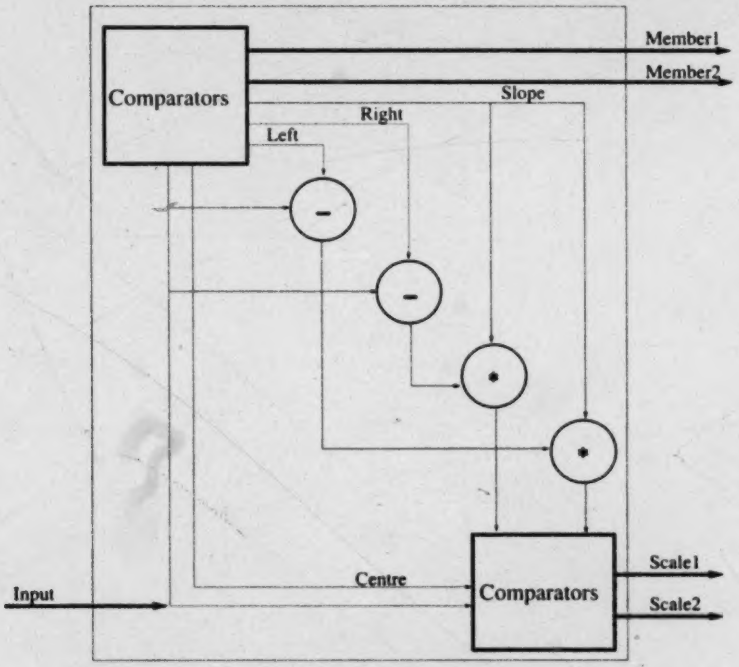


Figure 5.4: Detailed Schematic for Fuzzification

$$Total\ number\ of\ rules = \{Number\ of\ Fuzzy\ Sets\}^{\{Inputs\}} \tag{5.6}$$

Thus, the total number of slices utilized (T_{Slices}) increases exponentially with the increase in the number of inputs to the fuzzy controller.

$$T_{Slices} \propto Total\ number\ of\ rules \tag{5.7}$$

$$\propto \{Number\ of\ Fuzzy\ Sets\}^{\{Inputs\}} \tag{5.8}$$

5.1.2.2 Block RAM implementation

Another technique of implementing the rule base is by using the existing Block RAMs in the FPGA. The memory address locations are defined by the combination of numbers identifying all the input Fuzzy Sets (membership functions). These memory locations store locations of the output Fuzzy Sets (membership functions). Hence, the combination of the input membership functions defines the depth of the memory block and the possible output member locations define the width of the memory required. The total memory required for the Fuzzy rule base is defined by

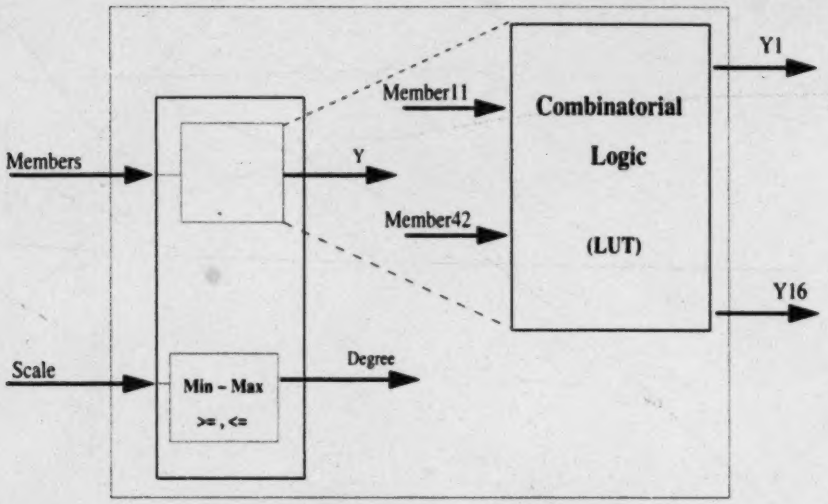


Figure 5.5: Schematic for Inference Scheme

the membership functions at the input and bit-resolution of the output membership functions. This organization is shown in Figure 5.6.

In Figure 5.6, Width = $2^{O_{members}}$ and Depth = $2^{I_{members}}$. Where, $I_{members}$ is the number of bits required to represent all the possible input membership functions. Also, $O_{members}$ is the number of bits required to represent all the output membership function locations, which is defined by the resolution of operation.

For example, consider a system with of 7 input and output membership functions, with a 8-bit number resolution. The total number of addressable locations are 2^7. Each of these location stores an 8-bit number, which stores the centre-point of the 7 output membership functions. Thus the total memory required is given by, $2^7 * 2^3 = 2^{10} = 2K$ bits for all the possible rules.

This relation can be generalized as follows:

$$Size\ of\ memory = 2^{I_{members}} * 2^{O_{members}} \tag{5.9}$$

$$= 2^{I_{members}+O_{members}} \tag{5.10}$$

$$= 2^{Bitlength} \tag{5.11}$$

Where,

$$Bitlength = I_{Bits} + I_{MBits} + O_{members} \tag{5.12}$$

Where I_{Bits} are the number of inputs to the system. I_{Mbits} is the number of bits used to represent the input membership functions. $O_{members}$ is fixed to 3, for a 8-bit representation.

The Block RAM approach shows a direct influence on the size of memory required based on the membership functions. This number is shared between the

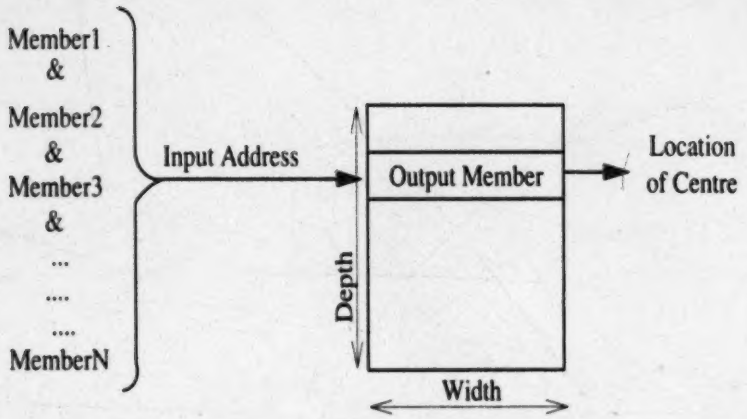


Figure 5.6: Memory Structure

total numbers of input fuzzy sets and the resolution of the output membersip functions. The memory size can be adapted to change this number by making suitable adjustment in the aspect ratio.

With this technique, there is delay of one clock associated with every rule. Hence a delay equal to the number of active rules is incurred. To minimize this delay, the copy of the rule base can be replicated for every active rule.

5.1.3 Defuzzifier

5.1.3.1 Centre of Gravity

The Figure 5.7 shows the logical implementation of the *Centre of Gravity* (COG) algorithm. As inferred from Equation 2.5, the hardware resources required are multipliers, adders and a divider. The multiplier computes the products and the adders compute the numerator and denominator. The divider performs the final division operation. This resource count (of adders, multipliers and the divider) increases exponentially as the number of active rules. The resource utilization may be expressed in the following relations

$$T_{Mult} \propto T_{Active\ Rules} \tag{5.13}$$

$$T_{Adders} \propto T_{Active\ Rules} \tag{5.14}$$

All the operations are combinatorial, except the division operation, which is synchronous. This operation incurs a delay equal to the width of the divider. Consequently, this operation is the main bottleneck in circuit operation.

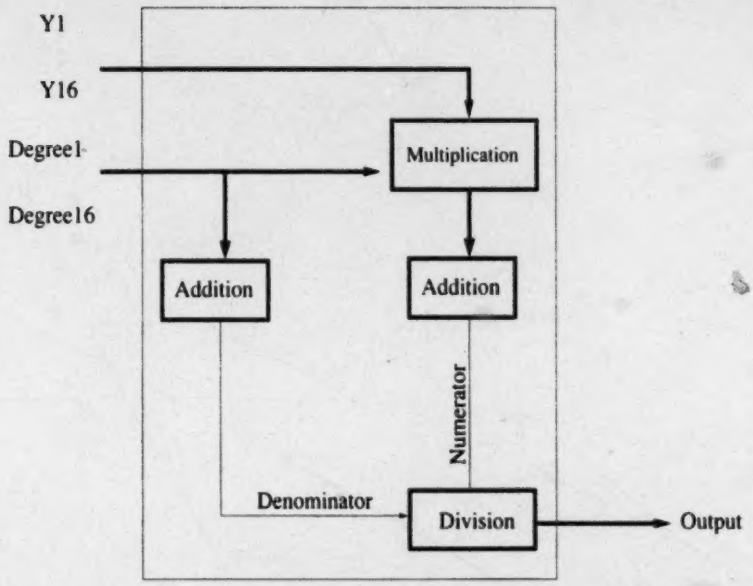


Figure 5.7: Schematic for Defuzzifier: Centre of Gravity

5.1.3.2 First of Maxima

The *First of Maxima* (FOM) method searches for the first location of the maximum value among the active rules evaluated. The hardware implementation of this type of defuzzifier requires comparators to search for the maxima. The number of active rules governs the number of the comparators required. Consequently the following relation gives the number of slices utilized,

$$T_{comparators} \propto T_{Active\ Rules} \tag{5.15}$$

Thus, area occupied by the defuzzification unit depends entirely the number of active rules, as in Equation 5.1.2.1.

$$T_{Slices} \propto T_{Active\ Rules} \tag{5.16}$$

5.2 Software Architecture

This section describes the implementation of the same functionality on a Microblaze soft processor. For this processor implementation, the following interface ports need to be included to provide external interface to the processor.

- Four input ports for the four crisp inputs
- One output port for the crisp output

5.2.1 IO configuration

External interface to the Microblaze can be configured in two possible configurations, viz., through general purpose IOs or by sharing the OPB bus among the external peripherals by defining dedicated ports through the user core template.

5.2.1.1 General purpose IOs

This section details the implementation technique using general purpose IO blocks available in the Xilinx EDK development environment.

Each *general-purpose IO* port (GPIO) is configured as an 8-bit wide interface as either inputs or outputs. The GPIO core is readily available in the EDK environment and connects to the OPB bus. The GPIO can be configured with a maximum width of 32 bits with two channels. It can be dynamically programmed as an input or output port. Also, to reduce resource utilization, the ports can be configured as dedicated input or output ports.

5.2.1.2 Fuzzy Logic Controller using General Purpose IOs

Using general purpose IOs to configure dedicated ports, four GPIOs configured as dedicated input ports and one GPIO as dedicated ports forms the interface to the design. Figure 5.8 shows the configuration of the software architecture, with a processor communication to the external inputs through the general purpose I/O ports.

With this configuration, the GPIOs each are 8 bit wide, allowing a maximum resolution of 2^8 to define the crisp inputs to the FLC. These five ports are connected to dedicated pins on the FPGA to allow data to be written or read continuously. Internally, the ports are shared on the OPB bus. The arbitration controller manages the sharing of the bus for read and write between these modules. The fuzzy logic controller code resides as a C code in the Microblaze processor memory. The processor addresses the location as defined by the general-purpose ports to access the crisp inputs and writes back to the ports after computation of the defuzzified values. The processor controls the operation of the fuzzification, fuzzy inference and defuzzification.

5.2.2 Hardware-Software Architecture

The Co-design architecture adheres to the structure defined earlier in Chapter 3. To implement this architecture, a system with a processor interfacing a dual port memory is required. The second port of the dual port memory interfaces with the design on dedicated hardware made of the FPGA resources. This structural implementation requires a user-defined core interfacing the dual port memory. The user-defined core internally consists of a BRAM interface controller and the logic defined on the hardware resources. This user defined core, interfaces with the on chip peripheral

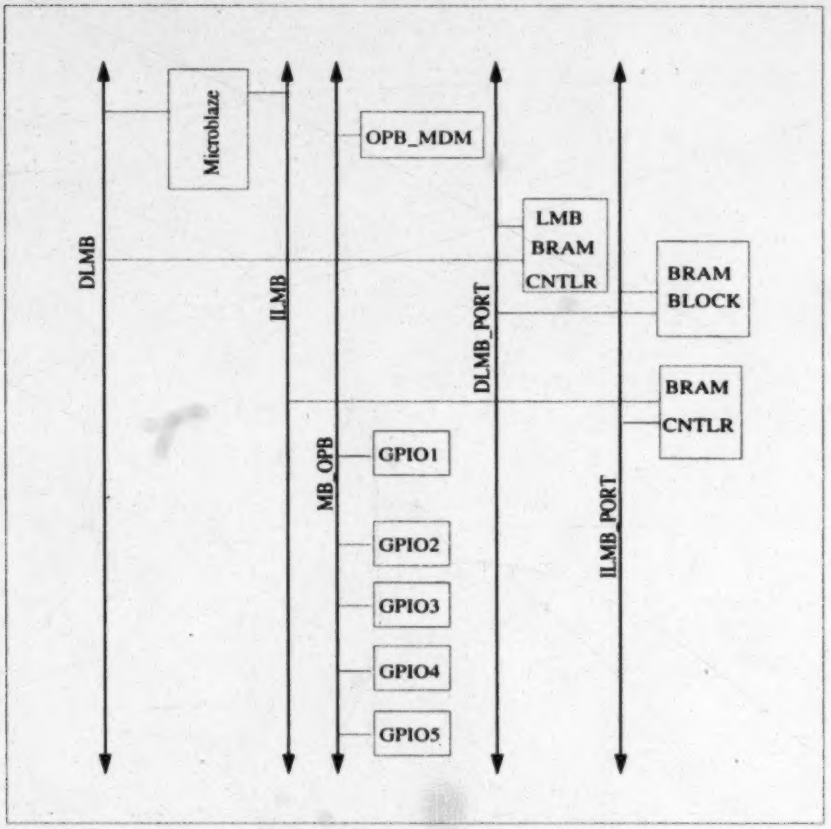


Figure 5.8: FLC with GPIOs

bus through a predefined bus interface called IP interface bus (IPIF). The hierarchical structure comprises of the OPB communicating with the user logic through the IP interface. The user IP interface internally comprises of the IPIF interface communicating to one of the ports of the BRAM controller. Since the BRAM block is configured as a dual port memory block, the second port can now interface with the logic defined on the FPGA resources.

5.3 Feasibility in Virtex-II: Hardware Architecture

The design environment used is the Xilinx Multimedia board with XC2V2000FF896 . The details of the board are provided in Section 6.6. In an attempt to quantify the feasibility of implementing a Fuzzy Logic Controller on an FPGA, we have tried to identify the upper bound or the largest design that could be accommodated on this FPGA.

From Equations 5.3, 5.4 and 5.5 for the fuzzifier, Equation 5.1.2.1 for the rule

evaluator and Equations 5.13 and 5.14 for the defuzzifier, it is observed that the resource utilization in a fuzzy logic controller increases exponentially as the inputs to the system.

5.3.1 FLC : Combinatorial Rule Base

Figure 5.9 shows the plot of the maximum number of inputs for a variation in the number of membership functions that can be accommodated in XC2V2000FF896. This implementation is based on the usage of distributed memory in the configurable logic blocks.

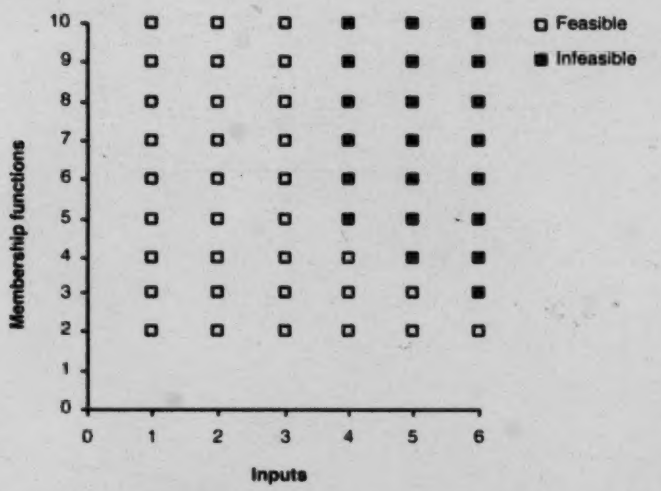


Figure 5.9: Feasibility: Combinatorial Implementation

5.3.2 FLC: Memory-based Rule Base

Table 5.1: Rule Base in Virtex-II

FPGA	Block RAM	Bit Length
XC2V2000	1008 Kbits	19
XC2V3000	1728 Kbits	20
XC2V4000	2160 Kbits	21
XC2V6000	2592 Kbits	21
XC2V8000	3024 Kbits	21

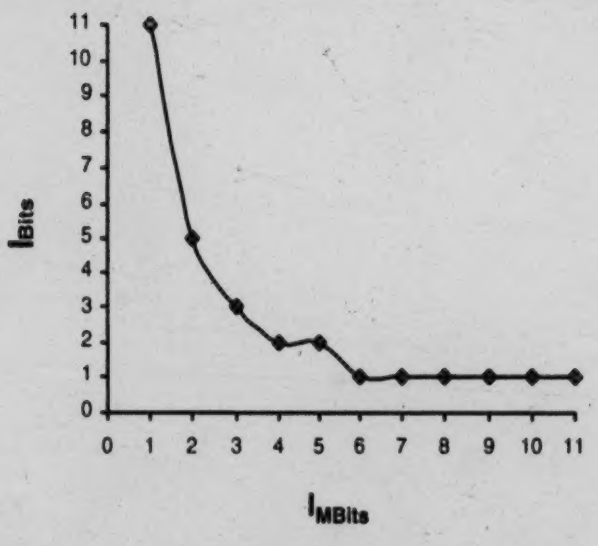


Figure 5.10: Feasibility: Memory based Implementation

The plot in Figure 5.10 shows the feasibility of implementing the maximum possible number of inputs on a given device. Table 5.1 represents the maximum memory size available in Virtex-II FPGAs and hence the maximum feasible rules. Bitlength is defined in Equation 5.12.

5.3.3 Resource Utilization

Table 5.2: FPGA Resource Utilization

Type	Module	Slices	LUTs	Gates	IOBs	BlockRAM
Combinatorial	FLC1: COG	72%	67%	98182	41	0
	FLC2: FOM	82%	71%	95907	41	0
Memory Based	FLC3: COG	10%	7%	2212324	41	57%
	FLC4: FOM	5%	5%	2139624	41	57%

Table 5.3 summarizes the distribution of resources, among the design modules in a 4-Input Fuzzy Logic Controller, with 5-output and 4-input triangular membership functions.

5.3.4 Frequency of Operation

The frequency of operation of a fuzzy logic controller is expressed in terms of *Mega Fuzzy Inference per second* (MFIPS). The following relation defines this:

$$MFIPS = \frac{Maximum\ frequency\ of\ operation}{T_{total}} \tag{5.17}$$

The maximum frequency of operation of the circuit is defined by the FPGA on which it is implemented and T_{total} is the total latency of the circuit. The total latency of the circuit accounts for the combinatorial delay at the fuzzification and rule evaluation stage and the pipelined delay of the divider. It is expressed in Equation 5.18.

$$T_{total} = T_{fuzz} + T_{rules} + T_{divider} \tag{5.18}$$

Table 5.3: Maximum Frequency of Operation

Module	Frequency (MHz)	T_{Total}	MFIPS
FLC1	30.20	16	≈ 2
FLC2	92.20	1	≈ 92
FLC3	49.42	32	≈ 1
FLC4	145.22	16	≈ 9

The T_{rules} for a synchronous Block RAM implementation is a one clock delay for every active rule. However, in a combinatorial implementation this delay is much smaller. The delay variations are summarized in Table 5.3.

5.4 Observation and Inference

Following are some inferences that can be drawn from these experimental results.

- The combinatorial implementation has very low latency, but the maximum frequency of operation and the largest design that can be accommodated is the lowest.
- Block RAM based implementation has a high latency, but also a significantly high maximum speed of operation.
- Area (Slices) occupied by the combinatorial implementation is higher compared to the Block RAM implementation on account of the CLBs, spread across the FPGA.
- The feasibility analysis considers all possible combination of rules. For all the feasible solutions, the speed of operation remains the same.

- The divider in the defuzzifier adds a significant amount of latency, also deteriorates the maximum operating frequency. Methods of pipelining the divider could improve the overall performance.
- The latency of the divider depends on its width, which is determined by the resolution of the numbers in the implementation.
- Although the number of rules is pre-determined for a fixed number of inputs and the associated fuzzy sets, limiting the design to a few rules can improve the speed of operation.

Chapter 6

Reconfigurable Fuzzy Logic Controller

In this chapter we look at the strategies adopted to build a Reconfigurable Fuzzy Logic controller. With a limitation on the resources and I/Os, we look at the largest configuration that can be accommodated on the FPGA available in the design environment. Next, the option of adapting partial run time reconfiguration to overcome these constraints is addressed. In the next section the technique of arriving at a self reconfiguring system in an attempt to design adaptive Fuzzy Logic Controllers, is discussed.

Following are some of the advantages of introducing reconfiguration in the design architecture.

- Adaptability to architectural changes
- Reusability
- Accommodate larger designs
- Dynamic design modification

6.1 Architectural Changes

For an adaptive architecture, it is essential for the design to be capable of rearranging itself in case of environmental changes affecting the architecture of the system. The impact of architectural changes for a Fuzzy Logic Controller is listed in the following sections.

6.1.1 System Inputs

The inputs to the system dictate the number of fuzzification units, the number of rules and consequently the number of active rules, (as seen from Equations 2.4 and

5.6), affecting all the modules in the design. A change in system inputs calls for a total reconfiguration of the FPGA.

6.1.2 Input and Output Membership functions

The number of membership functions associated with each input decides the input to the rule evaluator. Thus, a change in the number membership functions at the input changes the number of rules defined.

The number of output membership functions defined for the output of the FLC decides the contents of the rule base. Hence, a variation in the number of output membership functions will alter the rule base.

A reconfiguration of the rule base is necessary for a change in the number of output membership functions. By fixing a limit on the maximum number of membership functions that can be defined, the change in the number of membership functions can be integrated without any architectural change.

6.1.3 Defuzzification

A change in the defuzzification method calls for a re-design of the defuzzifier, which is an isolated module. Reconfiguring the defuzzifier can incorporate this design change.

Although the resource utilization varies with change in the method of defuzzification, the maximum number of inputs to the fuzzy logic controller that can be packed in an XC2V2000FF896 FPGA stays the same.

6.2 FPGA Configuration Mechanisms

The entire set of bits configuring the FPGA is called a configuration bitstream. These bits are grouped by columns, which are termed as frames. Loading the configuration frames into the internal configuration memory of the FPGA can configure the Virtex-II FPGA. This is a four-stage procedure, which involves, clearing the internal memory, initializing it by setting the mode of operation and then loading the configuration data. Finally the device is operated in the startup sequence of execution.

6.2.1 Configuration Modes

Dedicated pins on the FPGA, M0, M1 and M2 decide the mode of configuration, as shown in Table 6.1. The configuration modes of the FPGA can be classified as Serial and Parallel, based on the width of the configuration port. The following sections discuses the classification.

Table 6.1: Configuration Modes

	Modes	M0	M1	M2
Serial	JTAG	1	0	1
	Master Serial	0	0	0
	Slave Serial	1	1	1
Parallel	Master SelectMAP	1	1	0
	Slave SelectMAP	0	1	1

6.2.1.1 Serial Configuration Modes

The *Joint Test Action Group* (JTAG) has the standard IEEE 1149.1 for Test Access Port and boundary scan architecture. It complies with IEEE 1532, the standard for boundary scan based in system configuration of programmable devices. Dedicated pins TDI, TDO, TMS and TCK are used for this is serial mode of configuration. The JTAG pins are always available for configuration irrespective of the mode selected on the pins. However, the mode may be forced by explicitly making the selection on the mode pins.

In the Master and Slave Serial Modes, the FPGA receives the configuration data in a serial mode from an external serial *Programmable Read Only Memory* (PROM). One bit is loaded for every *configuration clock* (CCLK), with the *Most Significant Bit* (MSB) written first. In the Master serial configuration mode, the FPGA drives the CCLK, while in serial configuration; the external source drives the CCLK.

6.2.1.2 Parallel Configuration mode

The SelectMAP mode provides a bidirectional 8-bit configuration data interface. A data byte is loaded every rising clock edge of CCLK, which makes it the fastest mode of configuration. The master mode drives the CCLK and in the slave mode it is driven by an external source. With this mode of configuration, multiple FPGAs can be configured simultaneously. Once the process of configuration is complete, the SelectMAP pins can be re-used as user I/Os.

6.2.1.3 Comparative Speeds

Table 6.2 summarizes the relative speed of configurations for the various configuration modes.

6.2.2 Reconfiguration

Reconfiguration allows modifications to the existing design. The FPGA is reconfigured by reloading the internal configuration memory with a new bitstream, using one of the configuration modes. With this, part or whole of FPGA is cleared of

Table 6.2: Configuration Modes

Mode	Data Width (bits)	Operating Speed (MHz)	Download time (ms)
JTAG	1	33	206.13
Serial Mode	1	50	136.05
SelectMAP	8	50	17.01

it previous configuration and reloaded with the new design. The amount of time taken to reconfigure depends on the length of the bitstream. When the entire FPGA is reconfigured, it is termed as total reconfiguration or full reconfiguration.

Partial reconfiguration of the FPGA is supported only by boundary scan (JTAG) or Serial SelectMAP configuration modes. Only the bitstream which corresponds to the portion of the design to be reconfigured is directed to the FPGA. This approach is advantageous in terms of the reconfiguration time as compared to the case of total reconfiguration.

6.3 Partial Run time Reconfiguration in Xilinx FPGAs

The Virtex, Virtex-II , Spartan-II or Spartan-IIE family of FPGAs have the ability to reconfigure portions of the FPGA while the rest continues to function normally. This capability encourages further investigation of partial run time reconfiguration.

6.3.1 Module Based

Module based design involves partitioning the design into separate column-wise dedicated locations of the FPGA. This implies that any or all the functionality of these locations can be modified using this technique. However, the inter-modular communication has to necessarily remain fixed. Bus macros are used to isolate modular designs. This enables normal operation of the fixed modules during partial reconfiguration.

Module based partial reconfiguration has certain design considerations. Partially reconfigurable modules are necessarily selected column-wise for the entire height of the device. The minimum width is four slices and the maximum being the entire device width. All the resources that lie in this width encompass the 'frame' of the bitstream. The resources included are *Tristate Buffers* (TBUFS), slices, Block RAMs, multipliers, IOBs and the routing resources. Figure 6.1 shows the layout of the reconfigurable module. Bus macros are used for communication between these modules. The routing between these modules has to remain fixed. In Virtex-II FPGAs each row of CLB allows four bits of bus macro.

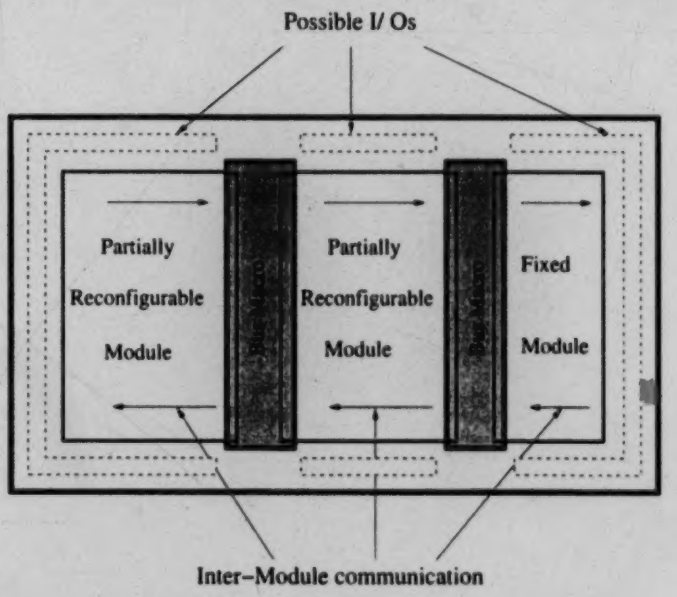


Figure 6.1: Layout of Partially Reconfigurable Modules, [5]

6.3.2 Difference Based

Difference based partial reconfiguration involves allows for modifications to the existing bitstream. These include modifications to the Look-up equations, re-writing the block RAM contents and changing the IOB characteristics. The changeable elements include MUXes, slices, Block RAMs, Flip-Flop initializations, and change in pull-up and pull-down elements that do not directly change the internal routing of the FPGA. These changes can be made directly using the FPGA Editor and then a new bitstream stream is generated which only includes the difference in the previous configuration file.

6.4 Reconfigurable Fuzzy Logic Controller on an FPGA

6.4.1 Module Based: Configuration 1

This configuration, shown in Figure 6.2, demarcates the Fuzzifier, Inference scheme and Defuzzifier as individual modules. This allows modifications within the module, while retaining the interconnection between the modules. In terms of fuzzification, this structure accommodates the change in the type of fuzzy membership functions (say triangular, trapezoidal or Gaussian). For the rule base, it allows re-defining the rules. With respect to defuzzification, this structure allows the algorithm to be partly or entirely changed. Further, the design change can accommodate

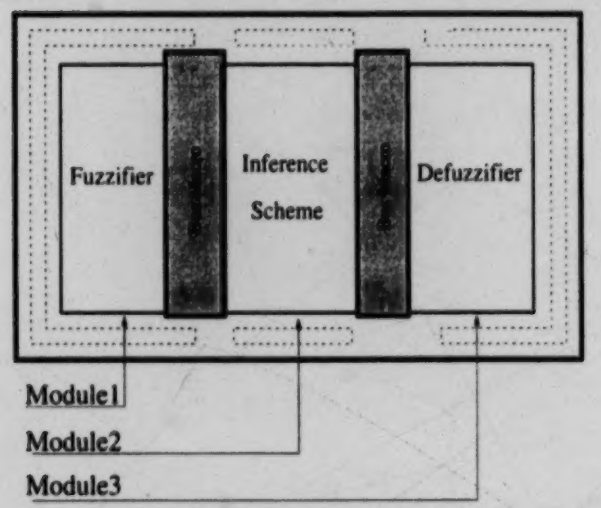


Figure 6.2: Configuration 1

any change in the number of bits required for defining inputs and membership functions to the system, while retaining the same routing to the inference.

6.4.2 Module Based: Configuration 2

Module 1 encompasses the fuzzifier and the rule base. The defuzzifier alone is partitioned as module 2, as shown in Figure 6.3. This configuration nullifies the interface between the fuzzification unit and the rule base. With this configuration, changes in the interface between the fuzzification and the rule base can be accomodated. However, the routing between the rule base and the defuzzifier is fixed. With this structure, changes in the number and type of input membership functions can be incorporated by reconfiguring module 1. However, the same number of active rules has to be retained for the fixed interface between modules 1 and 2. The defuzzification unit can be changed entirely.

6.4.3 Module Based: Configuration 3

This defines the entire design as a single module, allowing any and all possible combinations of modifications to be incorporated in the design. This essentially requires a complete bitstream to configure the entire FLC. This scenario exists, when the FLC itself forms a part of a larger design, shown in Figure 6.4.

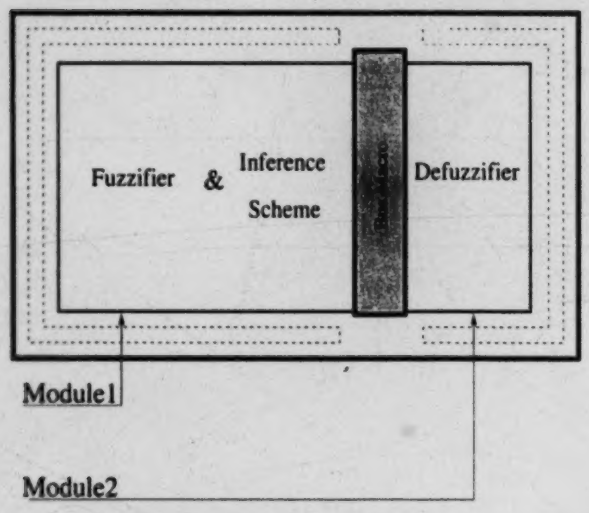


Figure 6.3: Configuration 2

6.4.4 Difference Based Reconfiguration

This technique is effective while making minor modifications to the original design. Some of the possible modifications that can be included are listed below.

- Membership function parameters: Like Slope and Co-ordinates.
- Type of membership function: For e.g. changing it from trapezoidal to triangular by equating *TRight* = *TLeft* (from Figure 5.2).
- Modifying the rules: Changing the rules in a memory based architecture
- Modifying weights applied rules: Nullifying a rule by adding a zero weight or adding a weight to activate a rule.

6.5 Reconfiguration Schemes

6.5.1 System ACE - CompactFlash Solution

The *Advanced Configuration Environment* (ACE) solution is an external reconfiguration scheme, which allows managing multiple bitstreams. The environment comprises of a compact flash memory module, an ACE controller and a microprocessor interface, as shown in Figure 6.5. The Compact Flash memory module can store one large bitstream or multiple bitstreams. The external interfaces to the ACE controller are detailed in the following sections, [6].

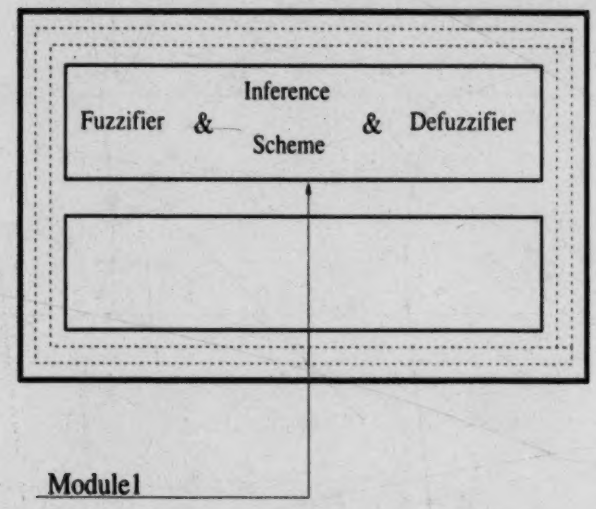


Figure 6.4: Configuration 3

6.5.1.1 Compact Flash Interface

This allows the controller to communicate to the Compact Flash. The interface is internally made up of an arbiter and a controller. The arbiter controls the access between the Microprocessor (MPU) and the JTAG ports. The controller detects the Flash memory module and allows read and write access.

6.5.1.2 Microprocessor Interface (MPU)

This interface monitors data access to and from the Flash memory module and is not required for normal functioning of the controller. This interface allows access to the internal registers of the controller. It is possible to alter the source of FPGA configuration bitstream through these internal registers. This allows dynamic reconfiguration by selecting bitstreams to be invoked as required from the CompactFlash

6.5.1.3 JTAG Interface

This is the dedicated configuration port, as described in section 6.2.1.1.

6.5.1.4 Active Modes

The following are the active modes of operation available for the FPGA configuration.

- *CompactFlash to JTAG setup*: This mode configures the FPGA directly from the CompactFlash on system Power ON through the JTAG port.

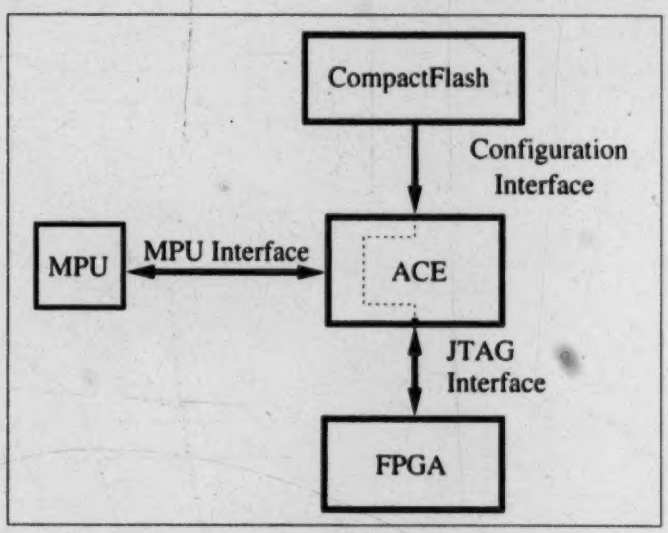


Figure 6.5: SystemACE environment,[6]

- *CompactFlash to MPU setup*: This mode allows altering the configuration start through the MPU interface. Once setup, this mode allows transfer of data from the CompactFlash to the MPU.
- *MPU to CompactFlash setup*: This mode sets up a communication path to the CompactFlash device through the MPU interface. This allows the contents of the CompactFlash to be accessible by the MPU.
- *MPU to configuration JTAG setup*: This setup allows the MPU interface to configure the FPGA via JTAG port.

The FPGA can be dynamically reconfigured based on the ACE controller settings. External inputs on pins CFGADDR (configuration address) dictate the address of the configuration file. Hence, changing the values on the CFGADDR pins and re-asserting the configuration reset (PROG) pin on the FPGA, can reconfigure the FPGA. The value of CFGADDR is controlled by external pins as well as the internal registers in the ACE controller. These internal registers are accessible through the MPU interface. The control register allows over-ridding the external pins and directs the CompactFlash to choose a different configuration file.

6.5.2 Internal Configuration Access Port

A *Internal Configuration Access Port* (ICAP) is located at the bottom-left corner in every Virtex-II and Virtex-II Pro FPGAs and allows access to the internal configuration memory, which allows in-circuit reconfiguration. This enables partial

reconfiguration of its own FPGA, enabling self-reconfiguration. This communication protocol is a subset of the SelectMAP interface and allows read-write access to all the configuration registers and memory. In the Xilinx Multimedia board (our board), the JTAG interface is hard-wired (M2=1, M1=0, M1=1), which disables the ICAP.

As ICAP allows access to the internal configuration memory of the FPGA, it is possible to reconfigure part or whole of the FPGA by directing the corresponding bitstream to this port. Consequently, if the design is modular and its location is fixed, it is possible to reload only parts of the design, while the rest of the FPGA functions normally. This technique allows runtime partial reconfiguration of the FPGA.

6.6 Design and Implementation on Xilinx Multimedia Board

The Xilinx Multimedia board has a Virtex-II FPGA, which can be configured through the JTAG port directly or through the CompactFlash System ACE interface. The board design limits the settings to the use of the JTAG interface, [42]

6.6.1 Default Configuration

The board design enables selection of bitstreams through an external 3-pin switch which is connected to the CFGADDR pins of the ACE controller. The current capacity of 16MB of flash memory can be shared between the eight configuration files. The default configuration mode picks the configuration file from the CompactFlash based on the CFGADDR pins and configures the FPGA through the JTAG interface.

6.6.2 Dynamic Reconfiguration

The MPU interface of the ACE controller is wired to the FPGA; hence the internal registers can be accessed only through the FPGA. The configuration mode used for implementation is shown in Figure 6.6

To enable access to the registers, a MPU interface is embedded within every configuration bitstream. Consequently a virtual MPU interface is embedded within the FPGA. This MPU interface allows read and write access to the registers, which allow over-ridding the external CFGADDR pins of the System ACE controller. Once the initial configuration is loaded into the FPGA, it is now possible to switch between configuration by writing into these registers and then initiating the reconfiguration of the FPGA. The embedded MPU interface is self-destructing. It initiates the process of reconfiguration, which reconfigures itself. It has to be noted, that this scheme only allows for the reconfiguration of the entire FPGA. This is on account of external method of reconfiguration, which starts the configuration process

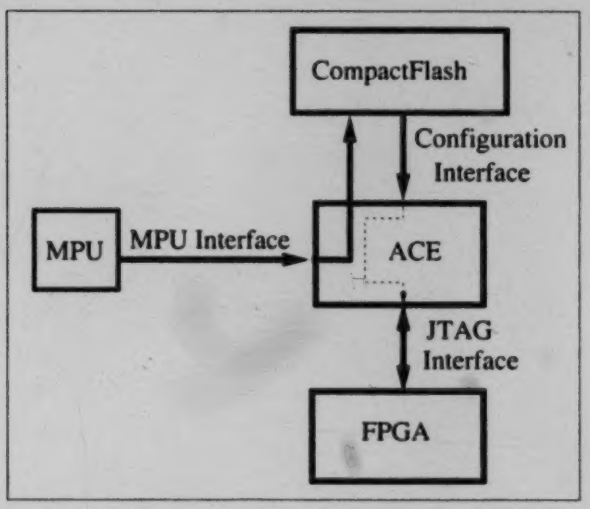


Figure 6.6: MPU setup, Configure from CompactFlash, [6]

by clearing the configuration memory of the FPGA. This method of configuration is shown in Figure 6.7 and Figure 6.8.

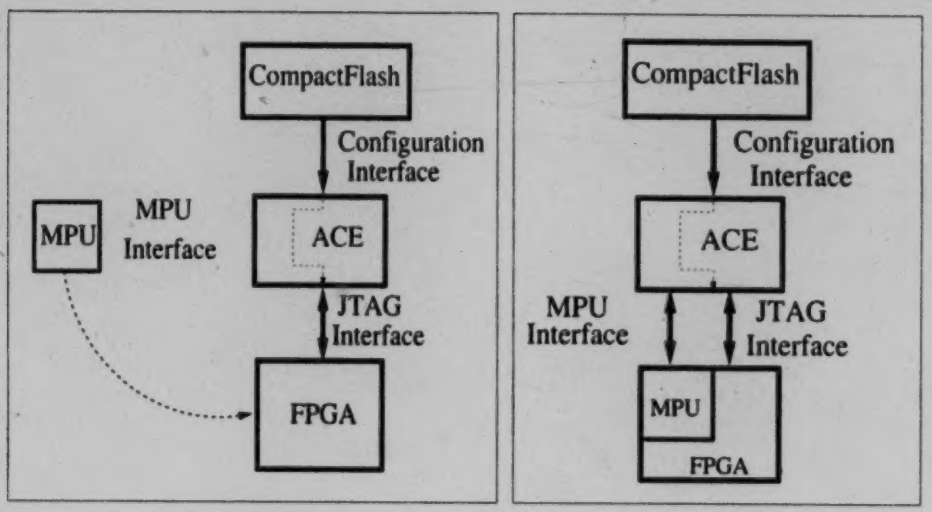
Using the SystemACE controller along with the ICAP enables remote runtime reconfiguration of the FPGA. In the modular design, the MPU forms the fixed logic, which controls the reconfiguration of the rest of the device. Since the ICAP does not differentiate between full and partial bitstreams, the device can be reconfigured without having to reset the rest of the FPGA or destroying the reconfiguration initiator. However, since the board is hard-wired to JTAG mode of configuration, the ICAP is disabled [43].

6.6.2.1 Internal Monitoring

Figure 6.9 illustrates the capability of internally monitoring the design, which includes the decision of invoking a reconfiguration of the design through the MPU interface.

6.7 Observation and Inference

- Minor design modifications using reconfiguration can be transparent to the circuit operation, by using partial runtime reconfiguration.
- Resource utilization in existing FPGA architectures can be extended through the application of self-reconfiguration and dynamic partial run time reconfiguration.



(a) Configuration 1

(b) Configuration 2

Figure 6.7: Configuration Change

- Capability of internal monitoring system can allow online-adaptability of the design.

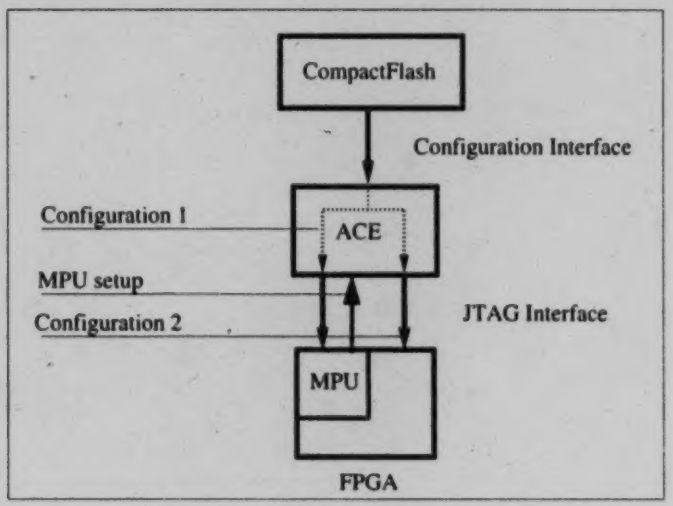


Figure 6.8: Sequence of configuration

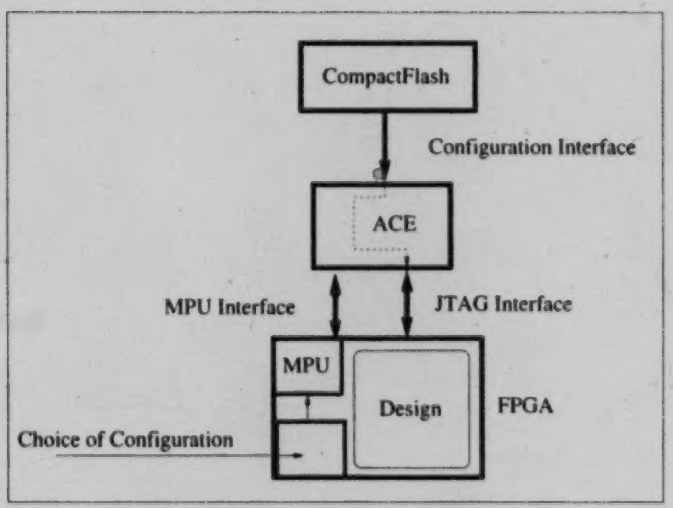


Figure 6.9: Partial Reconfiguration

Chapter 7

Conclusions and Future Work

The first section of this chapter outlines the goals accomplished while addressing the problem of Hardware-Software partitioning and the design strategies proposed for the implementation of a Reconfigurable Fuzzy Logic Controller. The second section suggests some areas of future research.

7.1 Conclusions

The contributions in terms of the design methodology adapted to implement a Fuzzy Logic Controller in a Hardware Software Co-design environment are summarized in the following sections.

7.1.1 Modeling

As a first step, a detailed model of the system architecture has been developed (see Chapter 3). This parametric model is used to study the performance of the system in terms of the execution time and power consumed. The model constructed is flexible and adaptable to different architectures specific to the application. The variations in the system architectures introduced by altering the timing parameters are illustrated in Appendix C.

7.1.2 Partitioning and Interfacing

In this thesis, an evolutionary-based approach performs design partitioning. The issue of partitioning has been converted to an optimization problem, which allows evaluation of the architecture with varying performance objectives. The focus is on the following design objectives: minimization of execution time, reduction of power consumption and the simultaneous optimization of the two. Furthermore, a procedure to optimize the resource allocation according to the design objective is presented.

Additionally, a heuristic technique which performs simultaneous scheduling and allocation is proposed. This algorithm performs a priority based assignment of time slots for every resource and simultaneously decides the task to resource mapping.

The process of communication between the two varying resources (software and hardware) is a critical issue - called interfacing. In this thesis, a simplified shared-memory-based architecture is introduced in a Co-design environment. This architecture avoids additional timing delays which can be incurred on account of bus arbitration in shared-bus architecture.

The evolutionary technique proposed shows a consistency in the quality of the results obtained, which establishes the dependability of the algorithm. The optimal resource allocation avoids under-utilization of the resources.

7.1.3 Validation

To confirm the techniques proposed, this thesis validates the optimization algorithm and the implementation architecture.

The evolutionary-based optimization procedure is verified by conducting multiple experiments for each of the performance objectives. The minimal variation in the results obtained for different runs validate the repeatability of the experiments (see Chapter 4).

Since the design involves both hardware and software architectures, it necessitates co-simulation. Due to the absence of a co-simulation environment, the hardware architecture and the software program were separately validated. The FPGA implementation is validated by performing both Functional and Place & Route Simulations in the Xilinx Design environment (ISE). The debugger in the Embedded Development Kit (EDK) allows validation for the software architecture.

7.1.4 Implementation

A reconfigurable Fuzzy Logic Controller is implemented using the Hardware-Software Co-design techniques developed in this thesis.

In Chapter 5, four design options are proposed, which is followed by a feasibility analysis. The largest design that can be implemented on a Virtex-II (XC2V2000FF896) device is estimated. The processing speeds for each of these designs, in terms of the Mega Fuzzy Inference per second (MFIPS), has also been inferred.

In Chapter 6, the software and hardware architecture of this application is described. A methodology to implement partial runtime reconfiguration of a Fuzzy Logic Controller is presented. This methodology uses the modular and difference-based design techniques, which permits reconfiguration at runtime.

The modular design strategies suggested allow incorporation of design changes transparent to the external interface.

7.2 Future Work

The focus of this thesis has been on merging the ideas of reconfigurable architectures and evolutionary algorithms. Firstly, using evolutionary techniques to achieve optimal hardware-software partitioning and secondly, in designing a reconfigurable Fuzzy Logic Controller. Following are some areas of possible future work proposed as an extension to the research work in this thesis.

7.2.1 Improvements in Hardware Software Partitioning

7.2.1.1 Two-level GA

A nested GA can be formulated on the hardware resources to arrive at the best possible combination of reconfigurable hardware resources. A similar second level of GA can be introduced to decide the resource-level granularity. The consistency in the results obtained in Chapter 4 shows that GA can be used as a tool for future design enhancements in hardware-software partitioning. Broadly, GA as a partitioning tool may be used in future to design adaptive system architectures by changing the task-resource mapping.

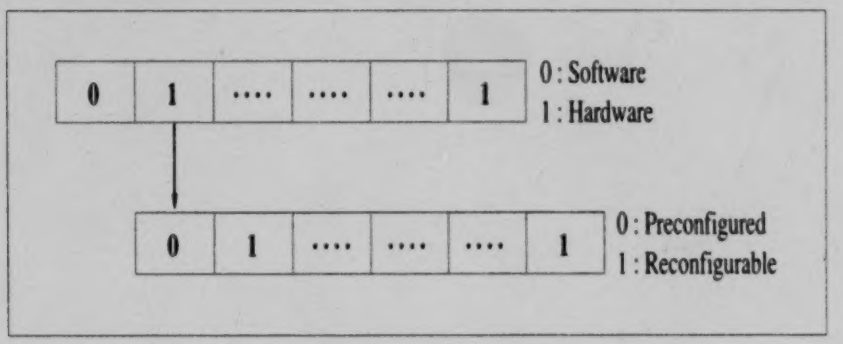


Figure 7.1: Two-level GA

Figure 7.1 shows the chromosomal representation of a two-level approach for system partitioning. The first level of GA identifies tasks that can be mapped on hardware and software. The second level of GA splits the tasks identified as hardware to preconfigured-fixed hardware and runtime-reconfigurable hardware. A similar approach could be applied to decide the granularity of the partition.

7.2.1.2 Self-Reconfigurable Hardware-Software Architecture

The system comprises of 'hardware' and 'software' resources and a control engine which performs the scheduling and allocation of the resources. The control engine

is basically a GA machine, which can be implemented either on 'hardware' or 'software.' This control engine is a 'fixed' module, which controls the reconfigurable modules.

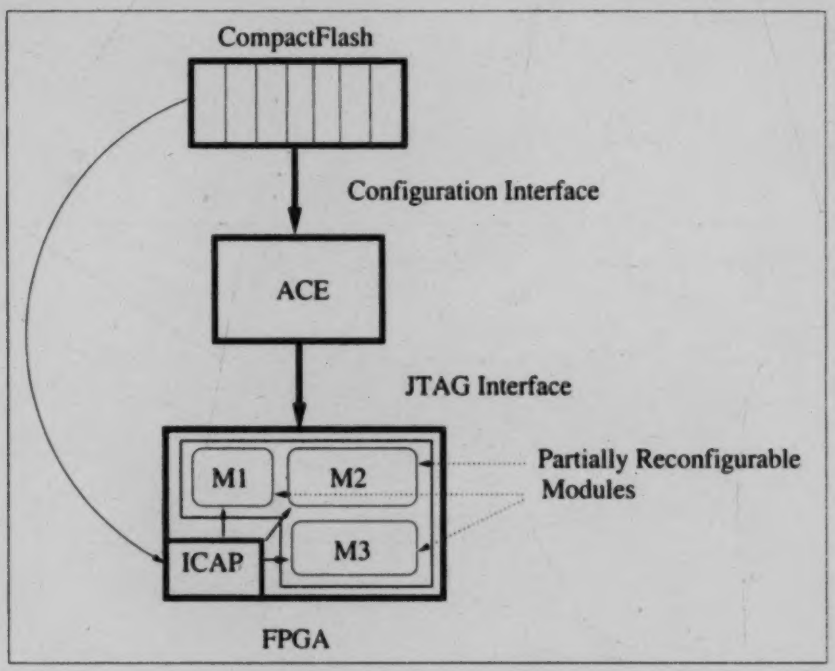


Figure 7.2: Partial Runtime Reconfiguration

As shown in Figure 7.2, an external memory unit stores the bitstreams for the hardware and software resources. These bitstreams are generated based on a modular design technique. The control engine 'calls' the resource according to the final results of the optimization algorithm. These bitstream are directed to the ICAP interface of the FPGA. The ICAP loads the bitstream to the internal configuration memory space, thus reconfiguring the portion of the device which requires modification, while retaining the functionality in the rest of the device.

7.2.2 Reconfigurable Fuzzy Logic Controllers: Design Alternatives

Figure 7.3 illustrates the design alternatives for a Hardware-Software combination of a reconfigurable Fuzzy Logic Controller. The fuzzification unit can be implemented on the processor or directly on the FPGA. The performance parameters of the fuzzification unit changes based on the type of implementation. According to the architecture under consideration (described in Chapter 3), there can be three

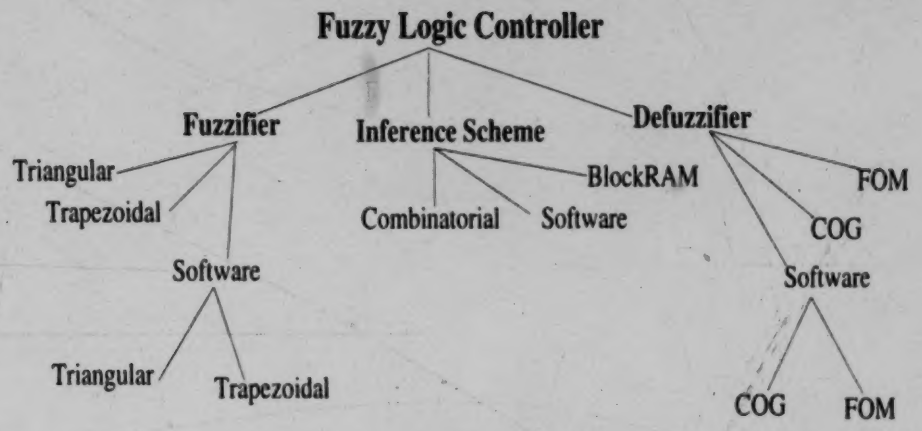


Figure 7.3: Design Alternatives

possible strategies of implementing each of the blocks of the Fuzzy Logic Controller. The fuzzifier can have triangular or trapezoidal membership functions and can have a circuit-level implementation or can reside on the processor. The inference scheme can be implemented in three different ways. Firstly, the option of implementing the inference scheme entirely on the FPGA using BlockRAM structures. Next, using the CLBs or lastly, using the embedded processor to set the rules using the internal memory of the processor. The performance parameters vary for each of the configuration. Similarly, the defuzzifier has two implementation options based on the target resource ('hardware' and 'software') or based on the algorithm of defuzzification (*Centre of Gravity* (COG) or *First of Maxima* (FOM), to name a few).

7.2.3 Hardware Software Co-Simulation

Commercially, tools supported for co-simulation (C and HDL) are limited. Programming languages such are SystemC [44] and Handel-C [45] provide the capability of building hardware architectures from a software C-like description. These ideas have further encouraged the approach of 'Computing without Processors' [46], which introduces the concept of reconfigurable devices for general purpose processing application. Future work in this area could propose a simulation tool, capable of validating the algorithm implemented in the co-design environment with the same ease as in a hardware (or software) environment.

7.2.4 Reconfigurable Processor Architecture

The approach of determining the optimal number of resources in terms of reconfiguration to allow maximum resource utilization can be extended to the design of a

reconfigurable processor. This allows the processor to be reconfigured according to the requirement of processing elements based on the target application.

Further, the process of optimization can be applied to arrive at the set of reconfigurable elements suitable for dynamically reconfigurable processors. Dynamically Reconfigurable Processors (DRP) is a mixed architectural approach which integrates general purpose computing and programmable logic.

7.2.5 Evolvable Hardware

Evolvable hardware is an approach to reconfigurable hardware design using methods inspired by biology. This concept differs from evolutionary circuit design due to the dynamic nature of the design environment. The major advantage of evolvable machines is its ability to evolve based on the information from the external environment.

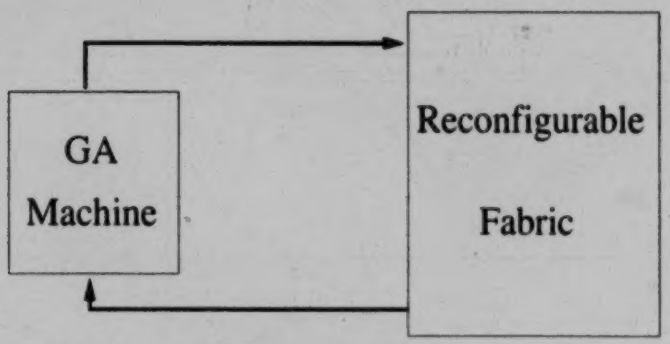


Figure 7.4: GA Machine

Figure 7.4 shows a high-level view of a self-evolving architecture. Such implementations have a built-in optimization machine. Here, GA is used to assist the design improvements. The fitness function acts as a feedback to aid the design to achieve its optimization goals.

A system with in-built optimization engine, allows adaptability to dynamic environmental changes. Implementing such systems needs investigation into the capability of sensing such variations and fast adaptations.

Bibliography

[1] Xilinx Inc., *Virtex-II platform FPGAs: Introduction and Overview*. Xilinx, 2004.

[2] ——, *Virtex-II platform FPGAs: User Guide*. Xilinx, 2004.

[3] X. Inc., *Microblaze Processor Reference Guide*. Xilinx, August 2004.

[4] N. Macias, "The PIG paradigm: The design and use of a massively parallel fine grained self-reconfigurable infinitely scalable architecture," in *The First NASA/DoD Workshop on Evolvable Hardware*, A. Stoica, J. Lohn, and D. Keymeulen, Eds., Jet Propulsion Laboratory, California Institute of Technology. Pasadena, California: IEEE Computer Society, 19-21 July 1999, pp. 175–180.

[5] D. Lim and M. Peattie, "Two Flows for Partial Reconfiguration : Module Based or Small Bit Manipulation," *Application Note : Virtex, Virtex-E, Virtex-II Pro Families*, May 2002.

[6] X. Inc., *System ACE CompactFlash Solution*. Xilinx, April 2002.

[7] C. Mandal, P. Chakrabarti, and S. Ghose, "Complexity of scheduling in high level synthesis," *VLSI design Journal*, vol. 7, pp. 337–346, March 1998.

[8] W. Wolf, "A Decade of Hardware/Software Codesign," *IEEE Computer*, vol. 36, pp. 38–43, January 2003.

[9] Wingtong, P. Cheung, and W. Luk, "Comparing three heuristics search methods for Functional Partitioning in Hardware-Software Codesign," *Journal of Design Automation for Embedded Systems*, vol. 6, pp. 425–449, July 2002.

[10] R. P. Dick and N. Jha, "MOGAC : Multiobjective Genetic Algorithm for Co-Synthesis of Hardware Software Embedded Systems," in *IEEE/ACM Confer-*

ence on Computer Aided Design, November 1997, pp. 522–529.

[11] T. Givargis and M. Palesi, "Multi-Objective Design Space Exploration Using Genetic Algorithms," *Proceedings of CODES 2002*, pp. 67–12, May 2002.

[12] M. López-Vallejo and J. C. López, "On the Hardware-Software Partitioning problem : System Modeling and Partitioning Techniques," *ACM Transactions on Design Automation of Electronic Systems*, vol. 8, no. 3, pp. 269–297, July 2003.

[13] G. Stitt, R. Lysecky, and F. Vahid, "Dynamic Hardware/Software Partitioning:A First Approach," *Annual ACM IEE Design Automation Conference*, pp. 250–255, 2003.

[14] V. Srinivasan, S. Radhakrishnan, and R. Vemuri, "Hardware software partitioning with integrated hardware design space exploration," in *Proceedings of DATE'98*, February 1998, pp. 28–35.

[15] R. P. Dick, D. L. Rhodes, and W. Wolf, "TGFF : Task Graphs For Free," *Proceedings of Int. Workshop Hardware/Software Codesign*, pp. 97–101, March 1998.

[16] D. N. Rakhmatov and S. B. Vrudhula, "Hardware-Software Bipartitioning for Dynamically Reconfigurable Systems," *Proceedings of CODES 2002*, pp. 145–150, May 2002.

[17] J. Harkin, T. M. McGinnity, and L. P. Maguire, "Genetic Algorithm driven Hardware-Software Partitioning for Dynamically Reconfigurable Embedded Systems," *Microprocessors and Microsystems*, vol. 25, pp. 263–274, August 2001.

[18] B. Mei, P. Schaumont, and S. Vernalde, "A Hardware-Software Partitioning and Scheduling Algorithm for Dynamically Reconfigurable Embedded Systems," *11th ProRISC Workshop on Circuits, Systems and Signal Processing Veldhoven, Netherlands.*, November 2000.

[19] A. P. Chandrakasan, R. Allmon, A. Stratakos, and R. W. Brodersen, "Design of portable systems," *Proceedings of the IEEE 1994 Custom Integrated Circuits Conference*, pp. 259–266, May 1994.

[20] J. Henkel, "A Low Power Hardware/Software Partitioning Approach for Core-based Embedded Systems," *Proceedings of Design Automation Conference 1999*, pp. 122–127, June 1999.

[21] G. W. Grewal and T. C. Wilson, "An Enhanced Genetic Algorithm for solving the High-Level Synthesis problems of Scheduling, Allocation and Binding," *International Journal of Computational Intelligence and Applications*, vol. 1, pp. 91–110, March 2001.

[22] T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, and R. F. Freund, "A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems," *Journal of Parallel and Distributed Computing*, vol. 61, pp. 810–837, 2001.

[23] C. L. Pape and P. Baptiste, "Heuristic control of constraint-based algorithm for preemptive job-shop scheduling," *Journal of Heuristics*, vol. 5, pp. 305–325, October 1999.

[24] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Inc, 1994.

[25] P. Crescenzi and V. Kann, "A compendium of NP optimization problems," *http://www.nada.kth.se/theory/problemlist.html*, August 1998.

[26] Z. Guo, W. Najjar, F. Vahid, and K. Vissers, "A Quantitative Analysis of Speedup Factors of FPGAs over Processors," *12th ACM International Symposium on Field Programmable Gate Arrays, Monterey California*, pp. 162–170, February 2004.

[27] J. Noguera and R. Badia, "Hardware-Software Codesign technique for Dynamically Reconfigurable architectures," *IEEE Transactions on VLSI systems*, vol. 10, pp. 399–415, August 2002.

[28] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.

[29] M. Gen and R. Cheng, *Genetic Algorithms in search and Engineering optimization*. MA: Wiley Interscience Publication, 2000.

[30] L. H. Tsoukalas and R. E. Uhrig, *Fuzzy and Neural Approaches in Engineering*. Wiley-Interscience, 1997.

[31] I.Baturone, A. Barriga, S. S. Solano, C. J. Jimenez-Fernandez, and D. Lopez, *Microelectronic Design of Fuzzy Logic-Based Systems*. CRC Press, 2000.

[32] D. G. Zrilic, J. Ramirez-Angulo, and B. Yuan, "Hardware Implementation of Fuzzy Membership Functions Operations and Inference," *Computers and Electrical Engineering*, vol. 26, pp. 85–105, January 2000.

[33] I. Kalaykov, "New Speed Limits for the Fuzzy Logic Controller Hardware," in *42nd Midwest Symposium on Circuits and Systems, 1999*, 8-11 August 1999, pp. 918 – 921.

[34] V. Salapura, "A Fuzzy RISC Processor," *IEEE Transactions on Fuzzy Systems*, vol. 8, pp. 781–790, December 2000.

[35] A. Gabrielli and E. Gandolfi and M. Masetti and M. Russo, "Design of a VLSI very high speed reconfigurable digital fuzzy processor," *Proceedings of ACM Symposium on Applied Computing, Nashville*, pp. 477 – 481, February 1995.

[36] Z. Salcic, "High-Speed Customizable Fuzzy-Logic Processor : Architecture and Implementation," *IEEE Transactions on Systems,Man and Cybernetics-Part A : Sytems and Humans*, vol. 31, pp. 731–737, November 2001.

[37] A. Gabrielli and E. Gandolfi, "A Fast Digital Fuzzy Processor," *IEEE Micro*, pp. 68–79, January/February 1999.

[38] T. Lund, A. Torralba, R. G. Carvajal, and J. Ramirez-Angulo, "A Comparison of Architectures for a Programmable Fuzzy Logic Chip," in *Proceedings of the 1999 IEEE International Symposium on Circuits and Systems, 1999 Volume: 5 , ISCAS '99*, 30 May-2 June 1999, pp. 623 – 626.

[39] T. Lund, A. Torralba, and R. G. Carvajal, "The Architecture of an FPGA-style Programmable Fuzzy Logic Controller chip," in *Proceedings 5th Australasian Computer Architecture Conference, 2000. Canberra, ACT , Australia*, February 2000, pp. 51–56.

[40] K. Singh, S.; Rattan, "Implementation of a Fuzzy Logic Controller on an FPGA using VHDL," in *22nd International Conference of the North Amer-*

ican Fuzzy Information Processing Society, NAFIPS 2003, July 24-26 2003, pp. 110 – 115.

[41] D. Kim, "An Implementation of Fuzzy Logic controller on the Reconfigurable FPGA system," *IEEE Transactions on Industrial Electronics*, vol. 47, pp. 703–715, June 2000.

[42] Xilinx Inc., *MicroBlaze and Multimedia Development Board User Guide*. Xilinx, 2002.

[43] ——, *OPB HWICAP*. Xilinx, 2004.

[44] G. Arnout, "SystemC standard," in *ASP-DAC '00: Proceedings of the 2000 conference on Asia South Pacific design automation*. ACM Press, 2000, pp. 573–578.

[45] S. M. Loo, B. Wells, N.Frejie, and J. Kulick, "Handel-C for Rapid Prototyping of VLSI Coprocessors for Real Time Systems," *Proceedings of the Thirty-Fourth Southeastern Symposium on System Theory*, pp. 6–10, March 2002.

[46] S. C. Goldstein, "Computing without processors." in *ERSA*, 2004, pp. 29–32.

Appendix A

Parameters to generate TGFF

Following is the input file for the TGFF generator, for the benchmark circuits in our experiments.

tg_cnt 5
task_cnt 2001
prob_multi_start_nodes 1
task_degree 22
period_mul 1, 1, 1, 1, 1
tg_write
eps_write
vcg_write
table_labelCOMMUN
table_cnt 3
table_attribprice 8020
type_attribexec_time 5020
trans_write

Appendix B

Parameters for Power and Time calculations

Table B.1: Parameters for Time calculations

Parameter	Unit Time
T_{sw}	30
T_{hw1}	3
T_{hw2}	4
T_r	3
T_{hh}	1
T_{hs}	2
T_{ss}	1

Table B.2: Parameters for Power calculations

Parameter	Unit Power
P_{ss}	1
P_{sh}	2
P_{hs}	2
P_{hh}	3
P_{h1}	2
P_{h2}	3
P_s	1
P_r	6

Appendix C

Architectures with Variations in Design parameters

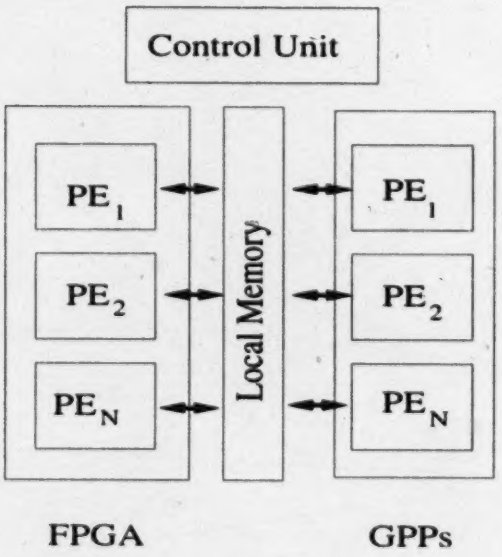


Figure C.1: Modified Architecture

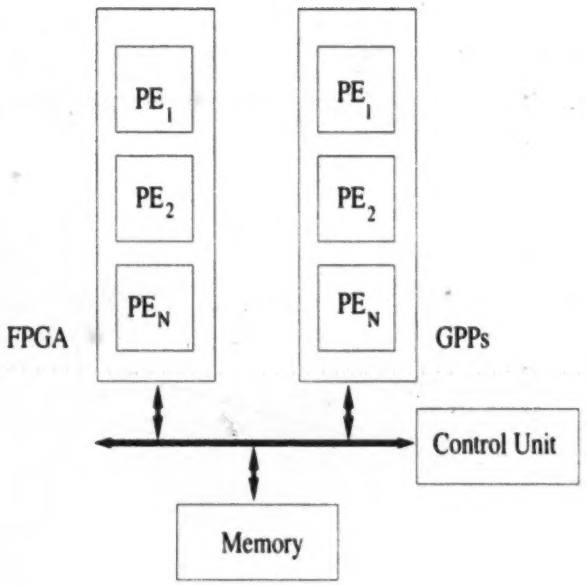


Figure C.2: Shared Bus Architecture